030-M098

AppleTalk Filing Protocol Engineering Technical Notes

A A P D A
Australasian Apple Programmers
and Developers Association

AppleTalk Filing Protocol (AFP) Engineering Technical Notes

Apple Computer Inc.

Protocol Version 1.1
February 17, 1987

Table of Contents

Contents

Chapter 1

AppleTalk Filing Protocol Design Description

This document specifies version 1.1 of the *AppleTalk Filing Protocol* (AFP). We start by discussing, in general terms, the scope of this protocol and some strategic decisions that have had a fundamental impact on its design. The relationship of this protocol to other parts of the AppleTalk protocol architecture are then spelled out. The detailed specification of the protocol is presented in three main parts: the AFP system model, AFP calls, and AFP packet formats.

The Basic File Access Model

The overall objective is to allow workstations on AppleTalk to access files on file servers connected to the network. This access occurs at the level of calls to the workstation's native file system. Figure AFP1 illustrates the basic file access model which can be used to make these general notions more precise.

As illustrated, a program running in a workstation issues *native file system commands*. If these commands refer to files on the workstation's local storage, then they are processed by the workstation's local/native file system. If however the commands refer to files on a file server, they are routed inside the workstation to a *Translator*. It is the Translator's responsibility to deliver the desired file system service to the requesting program. This is done by converting the native file system command into one or more AFP requests sent over the network to the appropriate file server. We can isolate two important file service interfaces in Figure AFP1.

The first, labeled *Native Filing Interface* (NFI), is the one through which programs on the workstation issue native file system commands and obtain the relevant file system services. A program "looking through" this interface sees the structures and capabilities of the workstation's native file system.

The second interface, labelled *AppleTalk Filing Interface* (AFI), is the one through which the Translator issues AFP requests to the file server and obtains AFP services. Looking through this interface, the Translator sees what we will call the AFP file system. It is the Translator's responsibility to map requests made through NFI into the requests that must be made through the AFI to the AFP file system in order to satisfy the original NFI request.

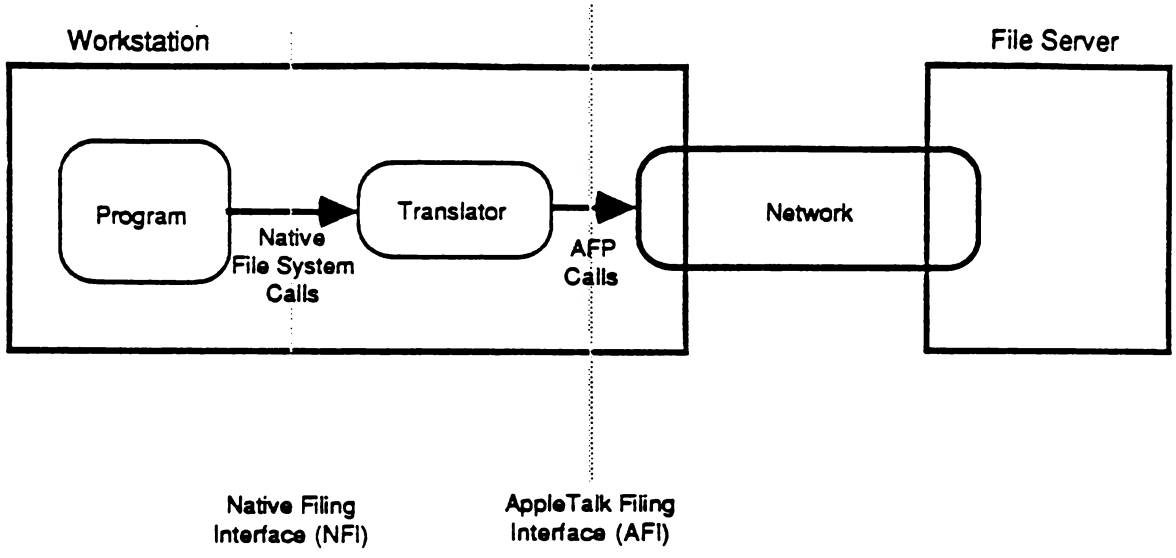


Figure AFP1. The Basic File Acce ss Model

Note that in some cases it may be necessary to provide another path from the Program directly to the AppleTalk Filing Interface to allow the Program to make AFP calls which have no equivalent in the Native Filing Interface.

The filing protocol specification in effect is the same as a complete specification of the AFI. This consists of three parts:

- the AFP file system structure;
- the AFP calls;
- the algorithms associated with these calls.

By the AFP file system structure is meant a complete description of entities such as servers, volumes, directories, files, forks, etc., that are "visible" through the AFI, their interrelationship and associated parameters. The client of the AFI (the Translator) sends calls/commands/requests through the AFI to manipulate this AFP file system structure, and the details of what each call does to this structure constitute the algorithms associated with AFP.

It is interesting to consider the relationship between the native file system and the AFP file system. Clearly, the latter must be functionally as powerful as the former. In fact, when the AFP is designed to allow various different types of workstations to use the model of Figure AFP1 at the same time, then the services provided by the AFP file system must, in a sense, be the functional union of the services provided by the various native file systems of these workstations.

This document does not examine in detail the mechanism of translation in the Translator; the focus here is on the AFI and the AFP itself.

Goals of AFP

We have paid special attention in the design of AFP to allow its extension in a very general fashion. This is essential if, in the future, additional types of workstations are to be supported by the protocol.

AFP version 1.1 is designed to work with Macintoshes (using the hierarchical file system) and MS-DOS workstations. Thus, the protocol is sufficient to successfully support translators on these two workstations under the indicated operating environment. We expect *future* extensions of the protocol to support the Apple-][(under ProDOS) and Unix workstations (but these are not part of the goals of version 1.1).

Access control mechanisms are an important part of filing protocols. AFP supports user authentication in a flexible fashion allowing the use of two standard password-based schemes but permitting future extension to other more sophisticated methods based, for instance, on public key encryption. AFP does not force the use of a fixed user-authentication method. AFP includes a directory-level access control mechanism based on user authentication at server login time. This scheme is similar to that used by Unix and therefore should be especially easy to implement on Unix-based server machines.

Although in the model of Figure AFP1 we have distinguished between workstations and file servers, AFP does not rule out the possibility of a particular network node being both a workstation and a file server. At the same time, AFP does not attempt to solve various concurrency problems and potential deadlock situations that can arise in such combined workstation/server nodes. This must be done by careful design of the software implemented on such "combined" nodes.

It should also be mentioned that AFP does not include any services or calls needed for administration of file servers. These will depend on the nature of a particular file server and are hence outside the scope of AFP.

AFP in the AppleTalk Architecture

As illustrated in Figure AFP2, AFP is a client of the AppleTalk Session Protocol (ASP) described in a separate document.

As should be clear from the ASP specification, a file server must call the AppleTalk Transaction Protocol (ATP) to open up a Session Listening Socket (SLS) and then use the Name Binding Protocol (NBP) to register the file service's name on this socket. Workstations wishing to use the file server must use NBP and the file service's name to discover the SLS's network address. With this information the workstation opens an ASP session with the server.

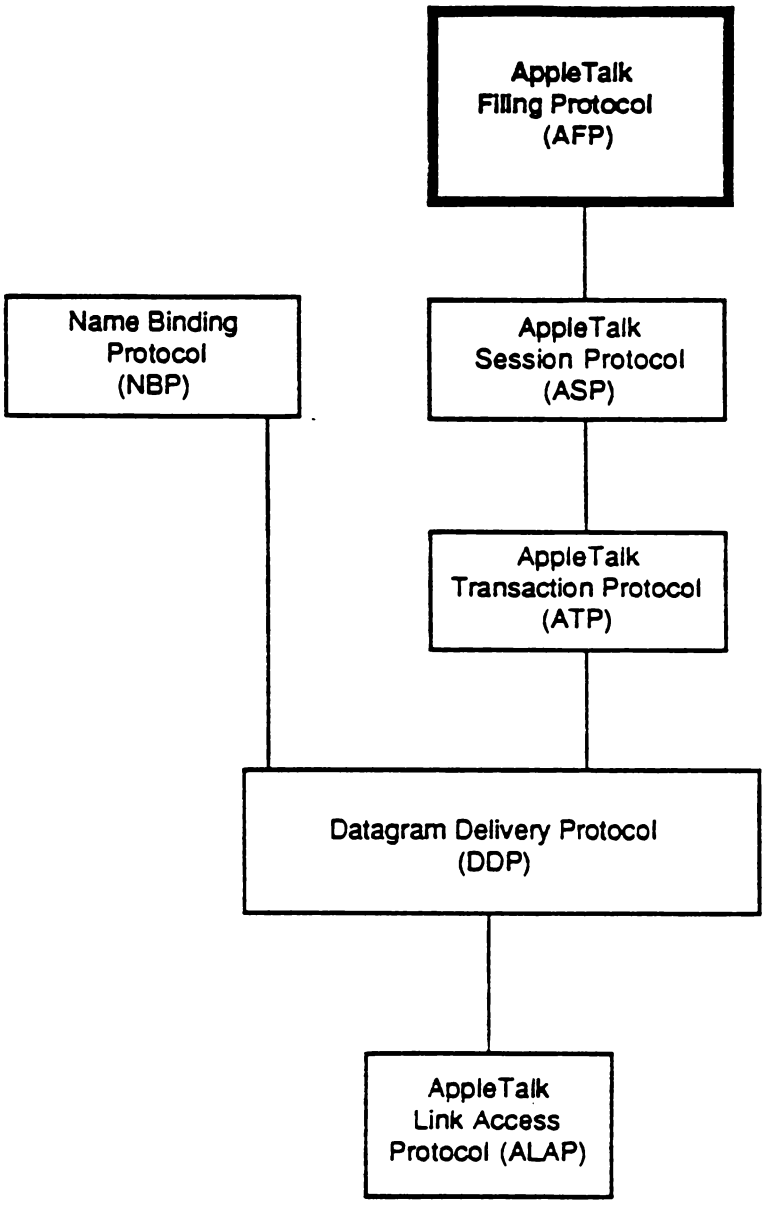


Figure AFP2. AFP and the AppleTalk Protocol Architecture

Once this session has been opened the AFP protocol entity in the workstation must log itself in on the file server as a bonafide user. The login step provides an opportunity to:

- authenticate the file server's user (i.e., the workstation);
- negotiate the version of the AFP protocol to be used during that session.

After the workstation has successfully logged in on the file server, the various AFP commands can be conveyed on the session thus established. When the workstation has finished using the services of the file server, it can logout. At this point all resources related to this filing session are freed up in the file server, and the underlying ASP session is closed.

Notation

Throughout this document hexadecimal (base 16) values are written with a leading $ sign (for example, $3A), while decimal integers are written with no leading special character (for example, 18).

Acronyms are widely used in this document (a list is provided in Appendix E).

The following abbreviations are used to describe the input and output parameters of AFP calls:

BIT	a single binary digit
BUF	a buffer; exact method of specifying location and length are dependent on the AFP implementation
BYTE	an 8-bit quantity
EntityAddr	a network-visible entity's internet address; exact size and format are dependent on the underlying network
INT	a 2-byte (16-bit) integer quantity
LONG	a 4-byte (32-bit) quantity
STR	a string consisting of a one-byte string length value (not including the length byte) followed by the string's characters (one character per byte). Strings cannot have more than 255 characters.
ResType	a 4-byte signature used in Finder info fields

Note that all string comparisons in AFP are case insensitive unless otherwise noted.

All numerical quantities are signed numbers unless otherwise noted.

Chapter 2

AppleTalk Filing Protocol System Description

This chapter contains a fairly detailed discussion of all the major concepts involved in AFP: the AFP file system structure, the login process, access control mechanisms provided by AFP, and a global discussion of the AFP calls.

Chapter 7 contains the detailed specification of the AFP calls. This forms the bulk of this document.

Although AFP has been designed to be used with various underlying transport mechanisms, on AppleTalk it is implemented as a client of the AppleTalk Session Protocol (ASP). In Chapter 8 we include a discussion of how AFP uses the services provided by ASP.

AFP File System Structure

Once the workstation client of AFP has logged into the server, it can issue any of a set of AFP calls described below. Through these calls the client can obtain descriptive information (collectively referred to below as parameters) about any of a set of entities that comprise the *AFP file system structure*, modify this descriptive information, create or delete entities, read and write to such entities, etc. Before discussing the calls for manipulating the AFP file system structure let us discuss this structure, its entities as seen through the AFI, and the relationship among these entities. [Note that this discussion does not describe how this structure might be implemeted on a server; it just describes the final result of such an implementation.]

The entities comprising the AFP file system structure are: [file] servers, volumes, directories, files, and forks. We now examine each of these and the relationship among them.

File Servers

The most global entity encompassed by the AFP file system structure and visible through the AFI is a *file server* (*server* for short). File servers possess names and can be discovered by a workstation as discussed below in connection with the login process. Servers manage one or more volumes which can be accessed through the AFP. (See Figure AFP3.)

There are several parameters associated with a server that are visible through the AFI:

- *server name* [string: max 32 characters]
- *server machine-type identification* [string: max 16 characters]

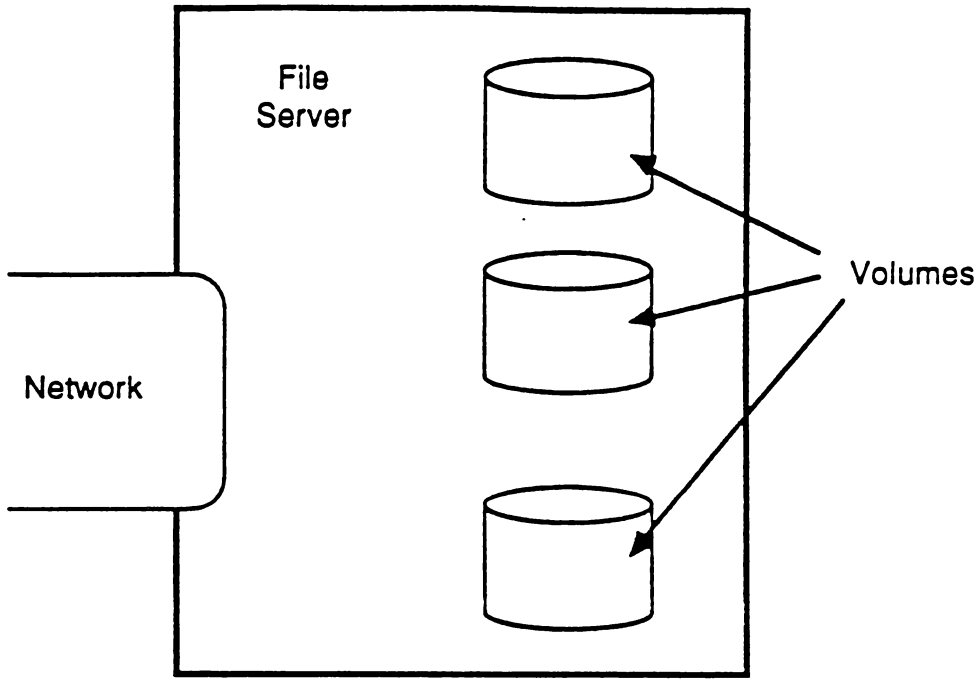


Figure AFP3: File Servers and Volumes

- *number of volumes* on the server [2-byte integer]
- *AFP version strings* [strings: maximum 16 characters per string]
- *UAM strings* [strings: maximum 16 characters per string]

In addition, a server can maintain the following optional parameter, which can be used to customize the appearance of a server's volumes on a Macintosh workstation's desktop:

- *server icon* [256 bytes].

The server icon consists of a 32-by-32 bit (128 bytes) icon bitmap followed by a 32-by-32 bit (128 bytes) icon mask. The mask usually consists of the icon's outline filled with black (a bit that is set). This format is exactly that required for the specification of icons for a Macintosh (see the *Structure of a Macintosh Application* chapter of *Inside Macintosh*).

The *server machine-type identification* is provided through a string of at most 16 characters. This string is purely informative, providing a textual description of the type of hardware and/or software system on which the file server has been implemented; it has no significance as far as the AFP is concerned.

The *AFP version strings* and the *UAM strings* will be discussed below in connection with the login process.

Volumes

As noted a server manages one or more *volumes* that are visible to workstations through the AFI. For each volume the server must maintain and make visible through the AFI the following parameters:

- *volume name* [string: max 27 characters]
- *volume signature* [2 bytes]
- *volume identifier* (VolID) [2 bytes]
- *volume creation date-time* [4 bytes]
- *volume modification date-time* [4 bytes]
- *volume backup date-time* [4 bytes]
- *volume size in bytes* [4-byte unsigned integer]
- number of *free bytes on volume* [4-byte unsigned integer].

In addition a server can maintain the following optional volume parameter which can be used to provide a simple kind of security at the volume level:

- *volume password* [8 bytes].

The *volume name* is a string of up to 27 bytes which identifies a server volume. The permissible characters in a volume name are all 8-bit characters not including null ($00) or colon ($3A). A given server's volumes must all have different names. The volume name is used only to identify the volume to the workstation's user. It is not used directly as part of the specification of objects on the volume (see the discussion of pathnames below). Instead, the workstation client makes an AFP call to obtain a particular volume's VolID, which is then used as the volume's identifier in all subsequent AFP calls.

The *volume signature* is a 2-byte quantity used to provide a volume-type identifier. In version 1.1 of AFP only three values of the volume signature are permitted (see the discussion below of volume signatures).

Within each filing session the server assigns a *volume identifier* (*VolID*) to each of its volumes. This value is unique (within a particular filing session) among the volumes of a given server and can be used in the AFP calls sent over a filing session to uniquely identify the volume to which the calls applies.

The *volume creation*, *modification*, and *backup date-time* quantities are maintained in terms of GMT values (see the discussion below). A volume's creation date-time is set by the server when the volume is created. Likewise, the modification date-time is changed by the server every time the volume is modified (see the *FPFlush* call). These two date-time values are managed solely by the server and they cannot be modified by the AFP client. However, the backup date-time is meant to be appropriately set by a backup program each time the volume's contents have been successfully backed up. When a volume is first created, its backup date-time is set to $80000000 (the earliest representable date-time value).

Directories and files are stored in volumes and constitute the next level of entities visible through the AFI.

The Volume Catalog: Directories, Files and Forks

A volume consists of directories and files arranged in a hierarchical structure known as the *volume catalog* (see Figure AFP4).

The *volume catalog* (also referred to as the *catalog*) is a description of the contents of the volume organized as a tree. The nodes of this tree, known as *catalog nodes* (*CNode* for short) are either files or directories. The *internal* (non-terminal) nodes of the tree are always directories, while the *leaf* (terminal) nodes are files or empty directories. At the base of each catalog is a special CNode (a directory) called the *root*.

A catalog does not span multiple volumes; the AFP client will see a separate catalog for each server volume visible through the AFI.

Directories should be looked upon as "logical containers" which contain other directories and/or files. Thus directories are equivalent to the concept of folders in the Macintosh user interface.

As in the Macintosh file system, a file consists of two *forks*: one, the *data fork*, is an unstructured finite sequence of bytes; the other, the *resource fork*, is typically used to hold Macintosh OS resources and a data structure for mapping them within the fork. As far as AFP is concerned both forks are simply finite-length byte sequences; specification of the resource structure of the resource fork is outside the scope of this protocol. Note that the bytes in a file fork are numbered starting with zero (the first byte in the fork is numbered 0).

A given file can have either or both forks empty. In fact, files created by an MS-DOS machine will most likely have an empty resource fork since this construct is outside the conceptual structure of an MS-DOS file and hence unintelligible to such a system. Likewise an MS-DOS machine accessing a server file created by a Macintosh will, in all likelihood, not access the resource fork of the file, and will in fact be unaware of its existence and significance. However, if applications are written for MS-DOS machines which understand and manipulate resource forks, they may do so; AFP does nothing to prevent this.

Non-Macintosh-based AFP clients that need only one file fork must use the data fork. It is important that the internal structure of the resource fork in terms of resources be maintained correctly. This is essential, since that is what a Macintosh workstation expects to see in the resource fork of any file. For this reason, workstations that do not know how to manage the internal structure of the resource fork should never alter its contents.

The tree structure of a catalog leads naturally to the idea of a CNode's *parent*, i.e., the node containing the given CNode. In terms of the pictorial representation of Figure AFP4, a given CNode's parent is the CNode immediately above it in the catalog tree. A CNode's parent will often also be referred to as its *parent directory*.

The set of CNodes contained in a given directory will be referred to as its *offspring*. This possibly empty set (for an empty directory) is exactly the set of CNodes for which the given directory is the parent. The *valence* of a directory indicates the number of offspring it contains.

Note that the root of the catalog is special in that it has no parent. For a given CNode *X* there is one path from the root to that CNode. Any CNode on that path is known as an *ancestor* of *X*. *X* is said to be a *descendant* of any of its ancestors.

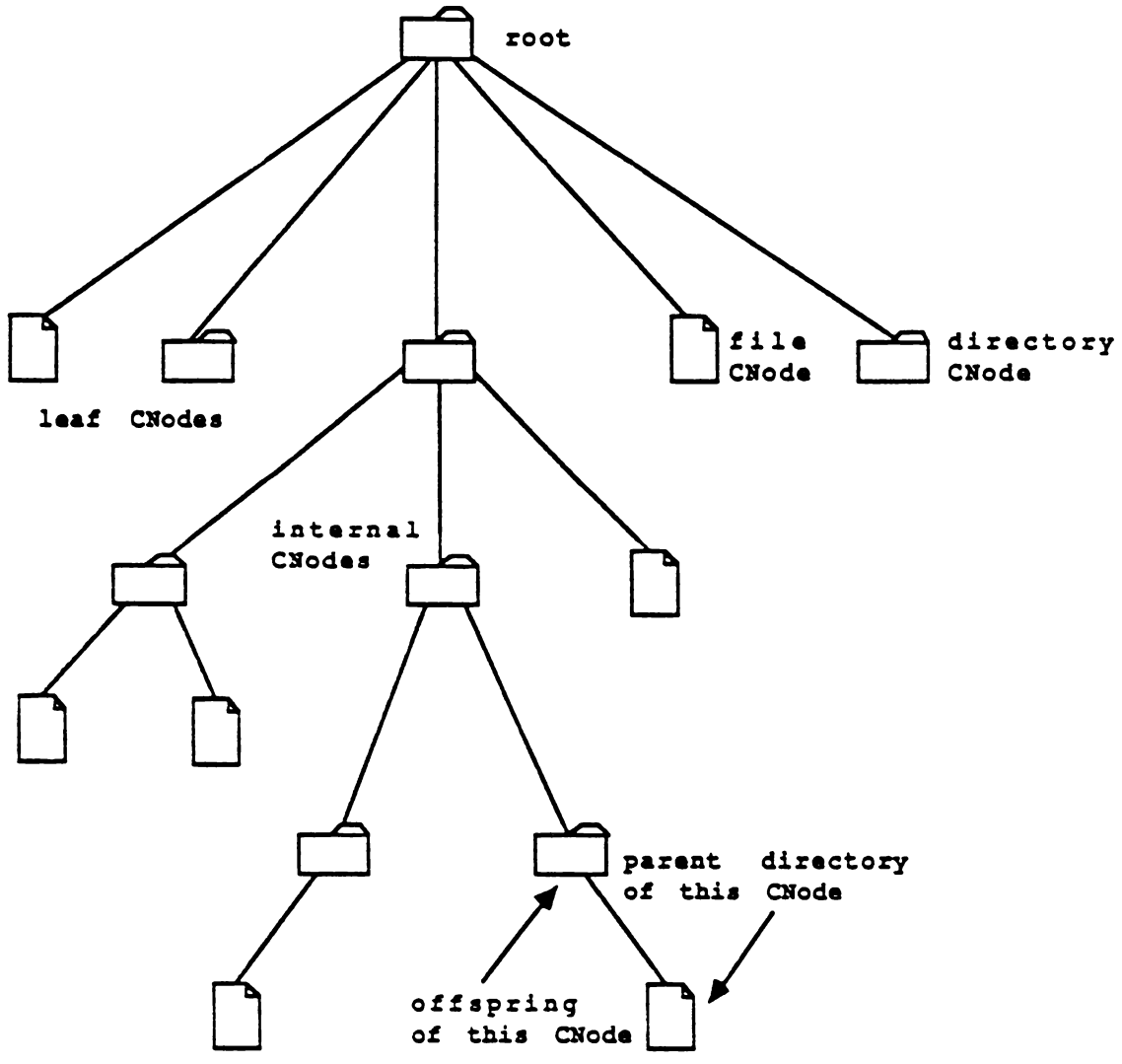


Figure AFP4: Volume Catalog

Catalog Node (CNode) Names: Long and Short Names

Every CNode (including the root) has two names: a *long name* and a *short name*.

Long names are strings of up to 31 characters. All 8-bit characters except for null ($00) and colon ($3A) are permissible characters for long names. With the exception of the null byte restriction, long names correspond exactly in syntax and significance to CNode names in the Macintosh's hierarchical file system (HFS). The mapping of character code to character is the same in AFP as in Macintosh.

Short names are strings of up to 12 characters that follow the well-known 8.3 format (<8 characters>.<3 character extension>) of file/directory names used by the MS-DOS operating system, and consist of a sequence of up to 8 characters followed by an optional

extension field consisting of a period ($2E) and up to 3 characters. All characters must be alpha-numeric or from the set ! # $ % &) (, - @ _ { } ~ (period is not included).

It should be clear why AFP attaches two names to each CNode, one each for the two native file systems supported by version 1.1 of AFP. Macintosh workstations expect to refer to files and directories by long names, while MS-DOS machines will use the 8.3 format short names. By using appropriate algorithms for creating the "other" name (long or short) when creating or renaming files and directories, a consistent sharing of these objects is possible from these dissimilar workstations.

The rules of uniqueness of long and short names are quite simple. No two offspring of a given directory can have the same short name or the same long name. A short name may match a long name if they both belong to the same object. Thus either name (long or short) uniquely identifies CNodes within the context of a given parent. To ensure that this uniqueness is maintained at all times, a file server must implement carefully-designed file/directory creation and renaming algorithms discussed in the specification section of this document and in Appendix B.

The root of the catalog is in fact a "container" representation of the volume. The root's long name is exactly the same as the volume name, so is therefore restricted to a maximum size of 27 bytes. The root of a volume can neither be deleted nor renamed through AFP. Hence a volume cannot be renamed through AFP.

Directory IDs

Each directory in the catalog has in addition to its names a numerical (4-byte) identifier known as its *directory ID* (*DirID* for short). The DirID of the root is always equal to 2.

Two directories on a given volume cannot have the same DirID. Thus the DirID uniquely identifies a directory on a given volume.

The DirID of a given CNode's parent is said to be the CNode's *ParentID*. This is a special case of the more general term *AncestorID*, which is the DirID of a CNode's ancestor. Although a given CNode has a unique ParentID (there is only one parent), it may have several AncestorIDs (one for each ancestor). The root's Parent DirID is always equal to 1. Zero is not a valid DirID.

Volume Signature

AFP allows three types of volumes. To determine what type a particular volume is, we associate a volume signature with each volume. This is a 2-byte quantity that can, for AFP version 1.1, take one of three permissible values:

01	Flat volume
02	Fixed DirID volume
03	Variable DirID volume.

Any value other than these should be taken as referring to a volume unintelligible to AFP version 1.1 and is hence an error condition.

A *flat volume* is one whose catalog tree consists of only one directory, the root, containing files. As an example consider the flat Macintosh file system's flat volumes. Attempting to

create a directory on a flat volume results in an error. The specification of files on such a volume consists of the VolID, a DirID having a value of 2, and the file name.

A *fixed DirID volume* is one that has a hierarchical catalog in which the directory IDs of the various directories in the catalog are fixed and determined at the time of creation of a directory. The DirID of a directory never changes during the lifetime of the directory. Furthermore, this DirID will not be used for any other directory during the lifetime of the volume (i.e., it is not used again even if the corresponding directory is deleted from the volume).

A *variable DirID volume* differs from a fixed DirID volume in that, although it maintains the uniqueness of DirIDs, it does not associate a fixed DirID with a directory over the entire lifetime of the directory. In fact, for such volumes the file server creates a unique DirID for a directory when the workstation client issues an FPOpenDir call (discussed in Chapter 7) and maintains this DirID value of a directory either until an FPCloseDir call is issued by the client or until the AFP session is terminated. Using a DirID obtained through an FPOpenDir call outside of this range of time will lead to unpredictable results.

Clearly, MS-DOS machines can access any of the three types of server volumes since the concept of DirIDs is foreign to their file system. However, it should be noted that the Macintosh using either MFS or HFS cannot directly use variable DirID server volumes (volume signature = 03). Macintosh HFS volumes are fixed DirID volumes and Macintosh applications such as the Finder save the DirIDs of various directories and expect these to remain invariant; hierarchical catalog volumes can be handled by this file system (HFS) only if they are fixed DirID volumes (volume signature = 02).

It should be noted that particular directories of a variable DirID volume can be mounted as flat MFS "volumes" and used by a Macintosh workstation. In this case, the corresponding "virtual" volume will be seen as flat in structure and any directories contained in that "volume"/directory will not be accessible through AFP. This is not a recommended way of implementing file servers for the Macintosh and its use is discouraged. In fact variable DirID volumes are included in our definition of AFP only to allow non-Macintosh machines (whose file systems are unable to implement, in an easy way, the fixed DirID feature) to function as simple servers offering files within particular directories for access over AppleTalk.

Directory and File Parameters

A server must maintain the following parameters for each directory:

- *long name* [string: max 31 characters]
- *short name* [string: max 12 characters]
- *DirID* [4 bytes]
- *ParentID* [4 bytes]
- *attributes* [2 bytes]
- *Finder Information* [32 bytes]
- *number of files and directories contained in the directory* [2-byte integer]
- *creation date-time* [4 bytes]
- *modification date-time* [4 bytes]
- *backup date-time* [4 bytes]
- *owner ID* [4 bytes]

- *group ID* [4 bytes]
- *owner's Access Rights* [1 byte]
- *group's Access Rights* [1 byte]
- *world's Access Rights* [1 byte].

The *Finder Information* is exactly the Finder information for directories that must be saved for HFS directories. This field is not examined by AFP. It is written (set) by the client of AFP. The last five fields in this list are related to access controls in AFP and will be discussed later in this document.

The *attributes* parameter is a bitmap indicating various attributes of the directory. For AFP version 1.1, one directory attribute is defined: *Invisible* (this directory should not be made visible to the workstation's user).

A server must maintain the following parameters for each file:

- *long name* [string: max 31 characters]
- *short name* [string: max 12 characters]
- *ParentID* [4 bytes]
- *file number* [4 bytes]
- *attributes* [2 bytes]
- *Finder Information* [32 bytes]
- *data fork length* [4-byte unsigned integer]
- *resource fork length* [4-byte unsigned integer]
- *creation date-time* [4-byte signed integer]
- *modification date-time* [4-byte signed integer]
- *backup date-time* [4-byte signed integer].

The *file number* is a 32 bit number associated with a given file, unique among all files on the volume. This is purely informative, since AFP does not allow the specification of a file by its file number.

The *attributes* parameter is a bitmap indicating various attributes of the file. For AFP version 1.1, five file attributes are defined. The rest of the eleven bits must be equal to zero. The five attributes are: *Read-Only* (cannot write to the file's forks), *DAlreadyOpen* (the file's data fork is currently open by some user), *RAlreadyOpen* (the file's resource fork is currently open by some user), *Multi-User* (the file is an application that has been written for simultaneous use by more than one user) and *Invisible* (this file should not be made visible to the workstation's user).

The *resource* and *data fork lengths* are equal to the number of bytes in the corresponding fork.

The *creation date-time* of a directory or a file is set to the value corresponding to the server's system clock when the file or directory is created. An AFP client with the appropriate access rights can set this date-time to any desired value. The *backup date-time* values are for the use of backup programs. When a file or directory is created the server sets the backup date-time to $80000000.

The *modification date-time* of a file is changed by the server every time either of the file's forks is closed or flushed, if this fork has been written to in that filing session (see the *FPClose* call). Furthermore, an AFP client with the appropriate access rights can set this date-time to any desired value.

The *modification date-time* of a directory is changed by the server every time the directory's contents are modified: this includes renaming the directory, creating or deleting a CNode in the directory, moving the directory, or changing its access privileges. Furthermore, an AFP client with the appropriate access rights can set this date-time to any desired value.

Date-Time Values

All date-time quantities used by AFP are Greenwich Mean Time (GMT) values. They are 32-bit signed integers corresponding to the number of seconds measured from 12:00 am on January 1, 2000 (the start of the next century corresponds to date-time=0).

The use of GMT makes these AFP date-time quantities independent of the geographical location of the servers and workstations using AFP. The Translator in a given workstation must, if it wishes to present local time-date values, carry out the appropriate conversion based on the geographical time zone and date (to allow for the changes related to Daylight Savings Time, etc.) applicable to the workstation.

Simple implementations of AFP workstations may wish to just obtain the server's time at login (using the FPGetSrvrParms call), calculate the offset between it and the workstation's clock, and then add or subtract this offset to all date-times sent to or received from the server.

Pathnames

As noted above, CNodes (files and directories) have long and short names, either of which can be used to uniquely identify a CNode within the context of its parent directory.

CNode names can be concatenated with intervening null-byte separators to form *pathnames*. Each element of a pathname must be the name of a directory, except for the last one which can be the name of a directory or a file.

The elements of a pathname can be long or short names; yet a given pathname cannot contain a mix of long and short CNode names. With a given pathname it is necessary to associate a *pathname type* which indicates one of two situations: the elements of the pathname are all short names (*pathname type* = 1), or all long names (*pathname type* = 2).

As the term signifies, a pathname can be used to traverse the catalog tree. The starting point of this path must be a directory which is separately defined (by its DirID). The pathname must be parsed from left to right to obtain each element which is used as the next node on the path. A valid pathname must proceed step by step from parent to offspring. The first element of the pathname is interpreted as an offspring of the starting point directory. A single null-byte separator preceding this first element is ignored.

Provision is made in the pathname syntax for ascending from a particular CNode on the path up to its parent. This is indicated by two consecutive null-bytes in the separator. Three consecutive null-bytes in a separator mean move up two levels in the catalog tree, and so on.

The syntax of an AFP pathname can be summed up in the following BNF-like form:

```
<Sep> ::= <null-byte>+

<Pathname> ::= empty-string |
        <Sep>*<CNode name>(<Sep><Pathname>)*
```

where we use the notation (a^*) to mean a sequence of zero or more (a)'s, while (a^+) stands for a sequence of one or more (a)'s. From this BNF-like specification of pathnames it should be clear that it is a concatenation of CNode names delineated by separators consisting of one or more null bytes. Pathnames may also start or end with a string of null bytes.

AFP does *not* allow the inclusion of the root directory in pathnames, i.e., in the commonly used terminology of pathnames, AFP does not allow the use of full pathnames. The equivalent of a full pathname is achieved in AFP by specifying the starting point of the pathname as the root (DirID = 2) and presenting the pathname as descending down from the root (but not including the root).

Pathnames sent in AFP packets are formatted as Pascal strings starting with a length byte followed by up to 255 characters of the pathname (each pathname element does not include a length byte). In addition the pathname type must be provided. A single null byte is used to indicate that no pathname is supplied; note that even in this case, a valid pathname type must be provided.

Access Paths and Open File Forks

To read or write the contents of the data or resource fork of a particular file, the AFP client in the workstation must issue a special call to open the particular fork of the specified file. This leads to the creation of an access path to that file fork, and subsequent read and write calls will be processed by reference to that access path. To allow for this the server must generate an access path identifier known as the *Open Fork Refnum*, unique among all open forks on a given session. Also associated with a particular access path is an *Access Mode Descriptor*, which indicates whether this access path (open file fork) was opened for (and allows) reading and/or writing through that path.

For each access path (open file fork) the server must maintain the following parameters:

- *Open Fork Refnum* [2-byte integer] (zero is invalid)
- *Access Mode Descriptor* [2-byte bitmap]
- *Which-fork identifier* [1 significant bit in a byte].

In addition, the server must provide a way of accessing the file parameters of the file to which this open fork belongs.

The *Access Mode Descriptor* is maintained by the server and is inaccessible to workstation clients of AFP. It is used by the server to indicate the access mode with which that access path is open.

Complete Specification of a Catalog Node (CNode)

In AFP a particular CNode (file or directory) can be unambiguously specified to the server by providing the following: VolID, AncestorID, pathname. This specification subsumes special cases such as the specification of a particular directory just by VolID, AncestorID = DirID of the directory, and a null pathname (length byte of the pathname string equal to 0).

Chapter 3

The AFP Login Procedure

This chapter discusses the process of discovering a file server and logging in to it.

This process consists of three steps: (i) discovering the file server, (ii) obtaining file server information, (iii) the actual log on step. Before discussing these three steps we need to establish two relevant concepts: AFP versions and user authentication methods.

AFP Versions

As noted in the introduction, the AFP has been designed to be extensible. One of the techniques used towards this end is to associate one or more version descriptors, known as the AFPVersion with the protocol. These are strings of up to 16 characters which uniquely identify a particular version of the protocol. For instance for the version defined in the present document the AFPVersion string is "AFPVersion 1.1".

When logging in to a server the workstation must first find out which versions of AFP the server can handle. Then the workstation client of AFP selects the version it wants used on the session and indicates this in the session opening request.

User Authentication Methods

As part of the login process, AFP optionally provides for user authentication by the server. This involves conveying to the server some information identifying and validating the user. This can be done in many different ways depending on the level of security desired.

The basic problem is to identify the user to the server (this can be done with a user name) and then verify that this is a valid user. This user verification/authentication step can be based for instance on establishing in some way that this user has a secret piece of information (we will call this a password).

For instance, the user can send the user name and the password in clear text to the server which compares them against its list of valid information. The problem with this method is that anyone listening on the network with a "peek" utility can pirate this user/password information and use it for accessing the server.

A more secure method would be to send the server just the user's name. The server then consults its data base of user authentication information and extracts from it the user password. Then it uses a scheme based on a random number modification/encoding by the password to verify if the user in fact knows this password. The password is never sent over the network; in fact the encoding is done in such a way that the password cannot be extracted from the information transmitted over the wire.

In the future we expect many other methods to be developed for authenticating users over a network. To make AFP extensible in this respect and to allow different servers to choose

to implement any or all of these user authentication methods AFP expects the workstation client to obtain a list of all the user authentication methods the server handles, and then select one of these methods for the login step. The identification of user authentication methods is done by a string of up to 16 characters known as UAM. For AFP version 1.1 we describe three standard methods identified by UAM = "No User Authent", "Cleartxt passwrd", and "Randnum exchange". Details of these three methods are discussed in Appendix A.

Discovering a File Server

As noted above, a file server implementing AFP must, upon being started up, open a session listening socket (SLS) and register its name on this socket. The name registered must have as its type part the string "AFPServer". [Note that server name string comparison is case insensitive].

Now a workstation trying to discover this or all file servers in the AppleTalk zone of interest must use NBP to lookup on the name =:AFPServer@<zone's name>. As a result of this call, the workstation client will receive, from NBP, a list of all active AFPServers in the zone together with the full internet address of the corresponding SLSs.

With this, the workstation has completed the discovery step of the login procedure. Note that so far there has been no need to open a session with any file server.

Obtaining File Server Information

After the SLS address has been obtained in the previous step, the AFP client in the workstation issues an FPGetSrvrInfo call to obtain a variety of server information including the list of AFPVersion strings and list of UAM strings for the versions and user access methods that the server can handle.

The Login Step

From the AFPVersion strings and UAM strings obtained in the previous step, the AFP client at this stage selects one AFPVersion string and one UAM string. These correspond to the AFP version and user authentication method the user wants the server to use. Then the user makes an FPLogin call to AFP to initiate the login step. The caller must provide the selected AFPVersion string and UAM string, and of course the internet address of the SLS of the desired file server. Further information may have to be provided depending on the user authentication method selected.

The FPLogin call causes the opening of an ASP session with the server and then an authentication of the user. If this proceeds to successful completion, then an AFP session is open with the server and futher AFP calls can be sent to the server over this session.

The server associates with the user a 32-bit User ID and one or more 32-bit Group IDs, indicating the user's membership in those groups. In addition, one Group ID may be specially marked as the user's Primary Group, to be described below.

Chapter 4

Access Control Mechanisms in AFP

Access controls are built into AFP in three ways: user authentication at server login, an optional volume level password when first accessing ("opening") a volume, and directory-level access controls based on user authentication at login.

User Authentication at Server Login

User authentication at server login time has already been discussed above. This is the first "line of defense" as far as access controls are concerned. A file server can be set up to turn away an unauthorized user at this level and thus maintain the privacy and security of the file server's volumes and their contents. It is important to note that the user authentication step is vital to the directory-level access controls discussed below.

Volume Passwords

A second level of access control is provided through volume passwords. A server can associate an optional fixed-length 8-character password with each volume it is making visible through AFP.

As will become clear in the discussion of AFP calls, in order to make reference to a server volume the workstation must use a VolID. This VolID is obtained by the workstation through an FPOpenVol call in which the name of the volume is supplied as a call parameter. If the volume has a password associated with it, then the workstation must supply this password together with the FPOpenVol call in order to obtain the VolID (and hence the ability to issue AFP calls that make reference to that volume).

Volume passwords constitute a very simple protection mechanism for simple servers that do not wish to implement the full-fledged directory-level access control mechanism. Of course it is not as secure a mechanism either.

Directory-Level Access Controls

AFP includes a directory-level access control mechanism that constitutes the most secure method provided in this protocol. Note that, as mentioned earlier, AFP does not support any file-level access controls.

The basic idea is that directories are containers that contain objects: other containers (directories) and files. A user that has been authenticated at the server login step can try to make four classes of accesses to the directory's contents:

- *see directories* contained within that directory
- *see files* contained within that directory

- *read the contents of an object* (or view its parameters) contained within that directory.
- *write the contents of an object* (or change its parameters) contained within that directory.

Permission to search a directory's contents allows the user to list the names and parameters of other files and/or directories contained in this directory. Access to the contents of an objects in a directory can be of two types: Read and/or Write. Setting (changing) the parameters of an object (file or directory) is considered as Writing to the object.

AFP includes mechanisms for restricting these classes of access (Search, Read, and Write access) at the level of each directory in the catalog tree. This is done as follows:

With each directory we associate an owner and a group of users. The owner is initially set to the ID of the user that created the directory; the group is initially set to the user's Primary group, if one exists. The Primary group is like any other group affiliation, except that it is the group that gets assigned to any new directories that the user creates.

AFP recognizes three different access rights: Search, Read, and Write. The file server must save, with each directory, three *Access Rights bytes*, corresponding respectively to the directory's owner, the directory's group, and the world. Each access rights byte is used as a bit map whose three least significant bits are used to encode the three different access rights allowed by AFP for the corresponding user (i.e. owner, group, or world). More specifically, each directory has associated with it the following five parameters:

- *owner ID* [4 bytes]
- *group ID* [4 bytes]
- *owner's Access Rights* [1 byte]
- *group's Access Rights* [1 byte]
- *world's Access Rights* [1 byte].

In addition, the server must maintain a one-to-one mapping between owner ID (user ID) and user name (a string of maximum length 31 characters), and between group ID and group name (also a string of at most 31 characters). AFP includes calls to allow users to map IDs to names and visa-versa.

The most significant 5 bits of each access rights byte must be zero (they are reserved for access rights extension in future versions of this protocol).

When a user logs in on a server, as part of the user authentication mechanism, various identifiers are retrieved from a *user data base* maintained on the server. These include the user ID (a 32-bit number unique among all server users) and one or more (the exact number is implementation dependent) 32-bit group IDs which indicate the user's group affiliations. One of these group IDs is special in that it represents the user's Primary Group. This number is assigned as the group ID of each directory created by the user. Assignment of user IDs, group IDs, and Primary Groups is an administrative function and is outside the scope of this document.

When this user tries to access a directory or its contents, the following algorithm is used by the server to extract the rights corresponding to that user (UARights) and that directory:

```
UARights := Directory's world access rights;
If (User's ID = Directory's owner ID) then
```

```
    UARights := UARights OR Directory's owner's access rights;
  If (any of User's group IDs = Directory's group ID) then
    UARights := UARights OR Directory's group's access rights
```

The OR operations in this algorithm are inclusive OR operations. Having given this algorithm for determining the UARights corresponding to a particular directory we can now examine in more detail what rights are required for various AFP operations. We use the following notation: S^A = "search access rights to all ancestors down to but not including the Parent directory", W^A = "search or write access to all ancestors down to but not including the Parent directory, **Sp** = "search access rights to the Parent directory", **Wp** = "write access rights to the Parent directory", **Rp** = "read access rights to the Parent directory".

Almost all operations require S^A access, which means the user can only access the contents of an object in a given directory if he has permission to Search every directory in the path from the root to the parent's immediate parent directory. The exact access permitted to objects in the directory is then determined further by **Sp**, **Rp** and **Wp** rights for the Parent directory.

Creating a File or a Directory: The user must have W^A plus **Wp**. Hard Create needs the same rights as deleting a file.

Enumerate a Directory: The user must have Search access rights to all directories down to but not necessarily including the directory being enumerated. To view its offspring that are directories, Search access to the directory being enumerated is required as well. To view its file offspring, Search access to the directory is not required, but the user must have Read access rights to the directory.

Deleting a File. The user must have S^A, **Rp**, and **Wp**. Furthermore, a file can be deleted only if it is not open at that time.

Delete a Directory: The user must have S^A, **Sp**, and **Wp**. Furthermore, a directory can be deleted only if it is empty.

Rename a File: The user must have S^A, **Rp**, and **Wp**.

Rename a Directory. The user must have S^A, **Sp**, and **Wp**.

Get (Read) Directory Parameters: All this requires are S^A plus **Sp** access rights.

Get (Read) File Parameters: The user must have S^A plus **Rp**.

Open a File to Read its Contents: A file's fork must be opened in read mode before its contents may be read. To open a file for read (and thus to be able to read from it) the user must have S^A plus **Rp**.

Open a File to Write its Contents: A file's fork must be opened in write mode in order to write to the fork. To open a fork which is currently empty (both forks must be of zero-length) to write to it, the user must have W^A plus **Wp**. To open an existing fork (either fork is of non-zero length) to write to it, S^A, **Rp**, and **Wp** access is required.

Set (Write) File Parameters: The user must have W^A plus **Wp** to set the parameters of an empty file (when both forks are zero-length). To set the File Parameters on a non-empty (either fork) file, S^A, **Rp**, and **Wp** access is required.

Set (Write) Directory Parameters: The user must have S^A, **Sp**, and **Wp** access to change a directory's parameters if the directory is non-empty. If the directory is empty, the user must have W^A plus **Wp** to change its parameters.

Move a Directory or a File: Through AFP a directory or a file can be moved from a source parent directory to a destination parent directory on the same volume. The user must have S^A rights to the source parent directory, W^A access to destination parent directory, plus Write access rights to both source and destination parents. Furthermore to move a file, the user needs in addition Read access rights to the source parent directory. To move a folder, Search access to the source parent directory is required in addition instead of **Rp**.

Modify a Directory's Access Rights Information: A directory's Owner ID, Group ID, and three Access Rights bytes can be modified only if the user is the directory's owner and then only if the user has W^A plus **Wp** or **Sp** access to the parent directory.

Copy a file (FPCopyFile): To copy a file, possibly across volumes managed by the server, the user must have S^A plus **Rp** to the source parent directory and W^A plus **Wp** to the destination parent directory.

Special Cases

OwnerID=0 means that the folder is "unowned" or owned by <any user>. The owner bit of the User Rights summary byte is always set for such a folder.

GroupID=0 means that the folder has no group affiliation; hence the group's access privileges (R, W, S) are ignored for such a folder.

Chapter 5
A Discussion of AFP Calls

Now we provide an overall discussion of the various calls provided by AFP and how they can be used to access a file server. A completely detailed specification of each call is available in Chapter 7. For the purpose of this discussion we classify the calls into various groups.

Server-Level Calls

A workstation client can use the following server-level AFP calls:

- ***FPGetSrvrInfo***
- ***FPLogin***
- ***FPLoginCont***
- ***FPGetSrvrParms***
- ***FPLogout***
- ***FPMapID***
- ***FPMapName.***

To begin, a workstation uses the Name Binding Protocol to discover the server's session listening socket's network address (we call this the *SAddr*).

Next, the workstation must obtain server information by using the *FPGetSrvrInfo* call. This is done without opening a session to the server and it returns a block of server information containing the following server parameters:

- the server's name
- a string identifying the server's machine type
- a list of the AFP versions the file server can handle
- a list of the various user authentication methods the server can handle
- an icon to be used for displaying server volumes on the Macintosh.
- a bitmap of flags.

After making this call the workstation's AFP client selects one AFP version and one user authentication method. Using this selection it makes the *FPLogin* call to establish a session with the file server. A session is needed before any of the other calls can be made to the server. This login step involves authentication of the user; it returns an SRefNum, a session reference number to be used in all calls made over this session. Depending on the chosen authentication method, the entire authentication process may involve additional *FPLoginCont* (Login continue) calls to provide more information to the server.

After a session has been established with the server, the workstation must obtain a list of the volumes on the server. This is done by making the *FPGetSrvrParms* call, which returns a count of the number of such volumes, the names of these volumes, and an indication of whether or not these volumes are password-protected.

When a workstation no longer needs to communicate with a server it issues an *FPLogout* call to terminate the session.

The *FPMapID* and *FPMapName* calls relate to access controls issues. The *FPMapID* call is used to obtain the User or Group Name corresponding to a given User or Group ID. The *FPMapName* call provides the inverse functionality, mapping a User or Group Name into the corresponding User or Group ID.

Volume-Level Calls

There are five volume-level AFP calls:

- ***FPOpenVol***
- ***FPCloseVol***
- ***FPGetVolParms***
- ***FPSetVolParms***
- ***FPFlush.***

After obtaining the volume names through the *FPGetSrvrParms* call, the workstation client must make an *FPOpenVol* call for each volume it wishes to have access to. If the volume has a password attached to it, it must be supplied at this time. The call returns the volume parameters asked for in the call, the key one being the *VolID*. This VolID is used in all subsequent calls to identify the volume to which those calls apply.

The VolID remains a valid identifier either until the session is terminated or until an explicit *FPCloseVol* call is made. This call invalidates the VolID.

After obtaining a volume's VolID the workstation client can, at any time, obtain the volume's parameters by making an *FPGetVolParms* call. Likewise, the volume's parameters may be changed through an *FPSetVolParms* call.

A request may be made to the server to write out to its disk any data and control structures pertinent to a particular volume. This is done by making an *FPFlush* call.

Directory-Level Calls

There are five directory-level AFP calls:

- ***FPSetDirParms***
- ***FPOpenDir***
- ***FPCloseDir***
- ***FPEnumerate***
- ***FPCreateDir.***

The *FPSetDirParms* call allows the workstation client to modify a directory's parameters. The *FPGetFileDirParms* call (discussed below) is used to obtain a directory's parameters from the file server.

The *FPOpenDir* call is used to "open" a directory on a Variable-DirID volume and retrieve its DirID, which can then be used in subsequent calls to enumerate the directory or access its offspring. For Variable-DirID volumes, this is the only way to retrieve the DirID (using a *FPGetFileDirParms* or *FPEnumerate* call to retrieve the DirID on such volumes will return an error).

Note that it is not considered an error to make this call for directories on Fixed-DirID volumes (the fixed DirID will be returned), but this is not the preferred method. Use *FPGetFileDirParms* or *FPEnumerate* instead. Directories on variable DirID volumes can be "closed" by making an *FPCloseDir* call, which invalidates the corresponding DirID.

The *FPEnumerate* call is one of the most important AFP calls. It is used to enumerate (i.e., list) the objects (files and directories) contained within a specified directory. In reply to this call the server returns a list of directory and/or file parameters corresponding to these objects.

Directories are created through the *FPCreateDir* call.

Each of these calls requires the specification of the directory to which the call applies. This specification, in general, consists of the VolID, a DirID (of the directory or of an ancestor), and (if this DirID is of an ancestor, then) a pathname down to the specified directory. Remember that the DirID of the root of a volume is always equal to 2.

File-Level Calls

There are three file-level calls:

- ***FPSetFileParms***
- ***FPCreateFile***
- ***FPCopyFile.***

The *FPSetFileParms* call is used to modify a specified file's parameters. The *FPGetFileDirParms* call (discussed below) is used to obtain a specified file's parameters.

A file can be created through the *FPCreateFile* call.

A file that exists on a volume managed by a server can be copied to any other volume managed by that server with the FPCopyFile call.

Combined Directory-File-Level Calls

AFP includes five calls that apply to files and directories:

- ***FPGetFileDirParms***
- ***FPSetFileDirParms***
- ***FPRename***
- ***FPDelete***
- ***FPMove.***

The *FPGetFileDirParms* call is used to retrieve the parameters associated with a given object. Using this call, one need not know in advance whether the object is a file or a directory; indication of its type is returned from the call. Likewise, the *FPSetFileDirParms* call is used to set certain parameters associated with a given object, even if one does not know in advance whether the object is a file or directory. This call only allows the setting of those parameters that are common to both types of object.

The next two calls can be used to respectively rename or delete files and directories. A file can be deleted only if it is not open; a directory can be deleted only if it is empty.

The *FPMove* call can be used to move a file or a directory from one parent directory to another on the same volume. At the same time the object moved can be renamed.

Fork-Level Calls

There are eight fork-level calls:

- ***FPGetForkParms***
- ***FPSetForkParms***
- ***FPOpenFork***
- ***FPCloseFork***
- ***FPRead***
- ***FPWrite***
- ***FPFlushFork***
- ***FPByteRangeLock.***

A fork's parameters can be read or modified by using the *FPGetForkParms* and *FPSetForkParms* calls.

The *FPOpenFork* call is used to open an existing file's fork (either one). This call returns an OForkRefNum which is used in all calls that refer to this open fork. This reference number stays valid until the fork is closed through the *FPCloseFork* call.

The four remaining calls can be used to manipulate a fork opened through a previously issued *FPOpenFork* call. The contents of the fork can be read by making *FPRead* calls. The client can write to a fork by using *FPWrite* calls. The *FPFlushFork* call makes the server flush (write out to its disk) any of that fork's data that is in the server's internal buffers.

To allow for shared use of a file's open fork, a client can lock ranges of bytes in that fork. If a client locks a particular byte range (through the *FPByteRangeLock* call) then that range of bytes is reserved for exclusive manipulation by the client placing the lock; other clients can neither read nor write within the locked range.

Chapter 6

A Design for The Finder's Desktop Management In a Network Environment

The Finder presents Macintosh user with a unique user interface centered around the use of icons to represent objects on a disk volume. To present this interface to the user the Finder makes use of a number of data structures separate from the File System's volume catalog, all of which are maintained as resources of various types in an invisible resource file called 'Desktop'.

The Desktop file is currently used to perform three separate functions:

1. To associate documents and applications with particular icons through its 'bundle' mechanism, as well as storing the actual icon bitmaps,

2. To locate the corresponding application when a user opens a document, and

3. To store the text of comments associated with files and directories as part of the information displayed by 'Get Info'.

The management of icons centers around the concept of a *bundle*, stored as a resource of type BNDL in an application's resource fork. The BNDL resource, which is identified with a particular FileCreator type can, among other things, refer to a number of FREF resources, which can in turn indicate that a file of a particular type should be displayed using a particular icon. Together, BNDLs and FREFs can be used to determine the icon to be displayed for a given file from the Creator and Type information stored as part of its Finder Info in the catalog.

In addition, the Desktop file is used on HFS volumes to hold a list of applications stored in subdirectories on the volume. The desktop contains an APPL resource which is used by the Finder to find an application to launch when a document is selected, given the document's FileCreator information. The APPL resource basically maps a particular Creator type to a list of applications that can open documents of the specified type.

Finally, the Desktop file is used as a repository for the text of comments associated with files and directories on the volume. Comments are retrieved when the user selects 'Get Info' for a file or directory, at which time the comment text can also be changed.

The use of the current Finder in a Network Environment (i.e. on File Server volumes) has proven unsatisfactory: resource files, such as the Finder's Desktop file, are ill-suited for sharing among multiple users on a single File Server.

A new mechanism has been designed to replace the Finder's direct use of the Desktop resource file on File Server volumes completely. Eventually, this mechanism could be used transparently for both local and remote volumes. The interface to this new Desktop mechanism is presented below.

Call Interface

The interface includes three groups of calls:

1. Icon calls (AddIcon, GetIcon, and GetIconInfo),
2. APPL calls (AddAPPL, DeleteAPPL, and GetAPPL),
3. Comment calls (AddComment, DeleteComment, GetComment).

Each call is mapped into a corresponding AFP command. The semantics of the AFP calls exactly parallel the semantics of the interface routines. Like FPWrite, FPAddIcon is special in that it requires a special intermediate exchange of packets to transfer the data block representing the icon bitmap.

In the descriptions of the individual calls in the following sections, the data type **ResType** refers to the 4-byte signatures that are part of every file's Finder Information. Each file is assigned a 'File Type' meant to be representative of the nature of the contents of the file (PNTG, TEXT, etc.) and a 'File Creator', which is a unique signature indicating the application which created the file (such as MPNT, MACA, etc.).

Before any Desktop calls can be made, the user must make an OpenDT call, as follows:

- **Function OpenDT**(VolID: **Integer**; **Var** DTRefNum: **Integer**): **OSErr**;

The file RefNum returned for the Desktop Database must be used on future calls to indicate the Desktop Database being referred to. If an error occurred on the call, the refNum returned will be zero.

When all Desktop operations have been completed, the user should make a CloseDT call (which takes a single argument, the DTRefNum) and returns an OSErr. This will free all resources allocated as part of the OpenDT call.

Icon Related Calls

- **Function AddIcon**(DTRefNum: **Integer**; FileCreator, FileType: **ResType**;
  IconType: **Byte**; IconTag: **LongInt**; BitmapSize: **Integer**; Bitmap: **Ptr**): **OSErr**;

AddIcon adds a new icon bitmap to the Desktop database. The FileType and FileCreator arguments (4 bytes each) specify the set of files this icon is associated with, while the IconType argument may indicate a specific kind of icon. Note that for a given FileCreator/FileType, there may be a number of icons available, each with a different IconType. The IconTag argument indicates a LongInt value to be associated with the icon which will be returned along with the icon bitmap when it is retrieved. This could be used as a timestamp, for instance, to associate the creation date of the application with the icons it exports. Finally, the Size and Bitmap arguments provide the actual bitmap in questions.

If an icon of the specified IconType already exists for the indicated FileCreator and FileType, **AddIcon** will replace the bitmap stored with the new Bitmap. An error will be returned if the size of the new bitmap is different from the size of the old bitmap.

- **Function GetIcon**(DTRefNum: **Integer**; FileCreator: **ResType**; FileType: **ResType**;
  IconType: **Byte**; **Var** Length: **Integer**; BitMap: **Ptr**): **OSErr**;

GetIcon retrieves the bitmap for a given icon, given its FileCreator, FileType and IconType. If an icon of type IconType of the specified FileCreator and FileType is available, it is returned. Otherwise, an ItemNotFound error is returned. The length argument used on input to indicate the size of the buffer pointed to by the BitMap pointer. When the call is completed it is overwritten with the actual size of the bitmap returned.

- **Function GetIconInfo**(DTRefNum: **Integer**; FileCreator: **ResType**; IconIndex: **Integer**; **Var** IconTag: **LongInt**; **Var** FileType: **ResType**; **Var** IconType: **Byte**; **Var** Size: **Integer**): **OSErr**;

GetIconInfo retrieves a description of an icon, given its FileCreator type and a numerical index. It can be used to determine the set of icons associated with a given application without knowing the FileTypes in advance. Successive calls with increasing values of IconIndex will return information on all icons associated with a given Creator type.

Application Related Calls

- **Function AddAPPL**(DTRefNum: **Integer**; FileCreator: **ResType**; DirID: **LongInt**; CName: **String**[31]; APPLTag: **LongInt**): **OSErr**;

AddAPPL adds an entry for the application specified by the DirID/CName under the indicated ResType. The APPLTag argument is an additional LongInt stored with the mapping information. There may be more than one application with same FileCreator ResType, although the DirID/CName should uniquely identify the file. The Tag information might be used to decide among many possible applications which one to launch for a particular document (if the tag of the creator were stored in the Finder information of the document, for instance).

- **Function RemoveAPPL**(DTRefNum: **Integer**; FileCreator: **ResType**; DirID: **LongInt**; CName: **String**[31]): **OSErr**;

RemoveAPPL removes the mapping information for a given application indicated by its DirID/CName. Note that while the FileCreator type must be specified to locate the entry, the application tag is not required to remove an application entry.

Note that it is the Finder's responsibility to add and remove entries for applications which are copied to the volume or deleted, respectively. For entries which are moved or renamed, the Finder should remove the entry before the operation and add a new entry with the updated information after the operation has been completed successfully.

- **Function GetAPPL**(DTRefNum: **Integer**; FileCreator: **ResType**; Index: **Integer**; **Var** APPLTag: **LongInt**; **Var** DirID: **LongInt**; **Var** CName: **StringPtr**): **OSErr**;

GetAPPL looks up an application given its Creator ResType. The index argument is used to enumerate all application mappings stored. Indices 1 through n will retrieve the 1st through nth application mapping stored which are accessible by the caller (i.e. to which the user has Search and Read access). Unless the caller wishes to implement a special selection algorithm over all available applications, a single call to get the first mapping should suffice to find an application which can be launched to open the selected document.

Comment Related Calls

• **Procedure AddComment**(DTRefNum: **Integer**; DirID: **LongInt**; CName: **String**[31];
CommentText: **String**[199]);

AddComment stores a comment string associated with a particular file or directory on the volume. Unlike icons, there can be no more than one comment associated with any file or directory. If **AddComment** is called for a file or directory which already has an associated comment, the existing comment is replaced.

• **Function RemoveComment**(DTRefNum: **Integer**; DirID: **LongInt**;
CName: **String**[31]): **OSErr**;

RemoveComment removes the comment associated with a particular file or directory. An error is returned if no comment was stored for the file or directory.

Note that while the Finder will call **RemoveComment** to remove comments for files or directories when they are deleted, it does not call GetComment, RemoveComment and AddComment whenever a file or directory is renamed or moved.

• **Function GetComment**(DTRefNum: **Integer**; DirID: **LongInt**; CName: **String**[31];
Var CommentText: **String**[199]): **OSErr**;

GetComment retrieves the comment associated with a particular file or directory. If a comment is stored, the comment text is returned. If no comment exists, an error is returned.

Chapter 7

Specification of AFP Calls

This section provides a specification of the various AFP calls.

For each call a brief description of the call is provided. This is followed by a list of the input and output parameters of the call. For each call, the underlying transport mechanism is assumed to return an *FPError* whose relevant values for the particular call are listed and explained. For all calls, if the call completes successfully, then the error code value *NoErr* is returned. This particular value of *FPError* is not included in the descriptions of the calls. Likewise, a *UserNotAuth* error can be returned from almost every call, but is not included in the call descriptions. This error indicates that the user has not yet been properly logged in. In addition, a *MiscErr* can be returned from almost every call. *MiscErr* is used by the server to map other errors that don't have an equivalent AFP Error (like error in reading a disk sector).

A description of the relevant algorithm pertaining to the call is also included with the discussion. The access rights required by each call are also specified.

Each AFP call is sent to the server in the form of a *Command* block, to which the server responds with the four-byte *FPError* plus a *Reply* block. For each call, the formats of the Command and Reply blocks are provided in pictorial form. The pictures are drawn so that the width of the rectangles is one byte. The bit numbering and byte ordering of multi-byte fields is performed according to MC68000 standards: in the following pictures, high byte always appears above low byte. How these blocks translate into network packets depends on the underlying transport mechanism used by AFP. It should be noted that some calls return an empty Reply block (in these cases this block is not shown).

The *FPWrite* call is special in that the data to be written is not included in the Command block but is expected to be passed to the underlying transport mechanism as a separate block.

Many of the calls require a bitmap to be passed, along with a block of parameters packed in "Bitmap order." This order is defined as follows: the parameter corresponding to the least-significant set bit in the bitmap is packed first, followed by the parameter corresponding to the next-more-significant bit, etc., ending with the parameter corresponding to the most-significant bit.

All numerical fields represent signed numbers unless otherwise indicated.

FPAddAPPL

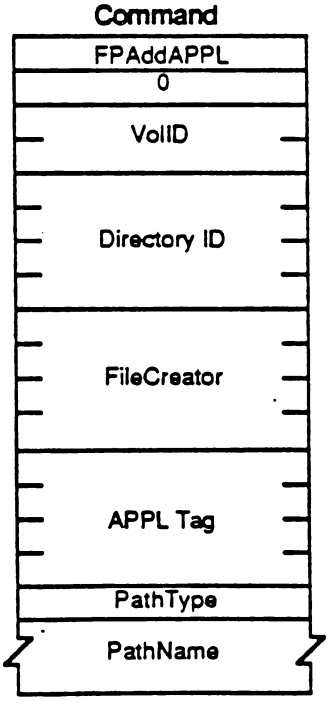
This call adds an APPL mapping to the Desktop Database.

INPUTS:	SRefNum (INT)	session refnum
	DTRefNum (INT)	Desktop Database refNum
	Directory ID (LONG)	ancestor directory identifier
	FileCreator (RESTYPE)	creator type of application being added
	APPLTag (LONG)	a user-defined tag stored with the APPL entry
	PathType (BYTE)	indicates whether pathname is composed of long names or short names: 1 = all pathname elements are short names 2 = all pathname elements are long names
	PathName (STRING)	pathname to the application being added
OUTPUTS:	FPError (LONG)	
ERRORS:	ObjectNotFound	input parameters do not point to an existing file
	AccessDenied	user does not have the rights listed below.

ALGORITHM: A mapping to the specified application is added to the Desktop Database. If an entry for the same application (same file name in the same directory with the same file creator) already exists, it is replaced.

RIGHTS: The user must have previously called FPOpenDT for the corresponding volume. In addition, the application must be present in the specified directory before this call is issued, and the user must have Search or Write access rights to all ancestors except the object's Parent, as well as Write access rights to the Parent.

PACKET FORMAT:



FPAddComment

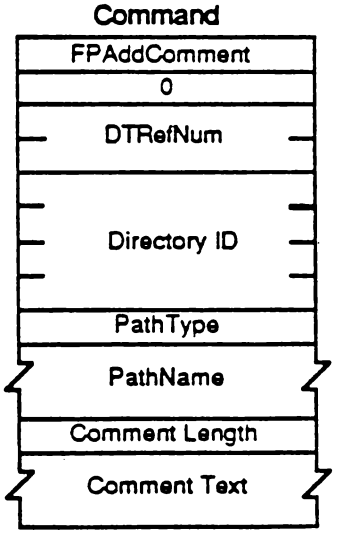
This call adds a comment for a file or directory to the Desktop Database.

INPUTS:	SRefNum (INT)	session refnum
	DTRefNum (INT)	Desktop Database refnum
	Directory ID (LONG)	directory identifier
	PathType (BYTE)	indicates whether pathname is composed of long names or short names: 1 = all pathname elements are short names 2 = all pathname elements are long names
	PathName (STRING)	pathname to the file or directory with comment
	CmtLength (BYTE)	length of comment data
	CmtText (BYTES)	comment data to be associated with file or folder specified (limited to 199 bytes)
OUTPUTS:	FPError (LONG)	
ERRORS:	ParamErr	unknown session refnum or Desktop Database refnum; bad pathname
	ObjectNotFound	input parameters do not point to an existing file or dir
	AccessDenied	user does not have the rights listed below

ALGORITHM: The comment type and comment data are associated with the specified file or directory and stored in the Desktop Database. If the comment length is greater than 199 bytes, the comment will be truncated to 199 bytes and no error will be returned.

RIGHTS: The user must have previously called FPOpenDT for the corresponding volume. In addition, the object must be present in the specified directory before this call is issued, and the user must have Search or Write access rights to all ancestors except the object's Parent, as well as Write access to the Parent.

PACKET FORMAT:



FPAddIcon

This call is used to add an icon bitmap to the Desktop Database.

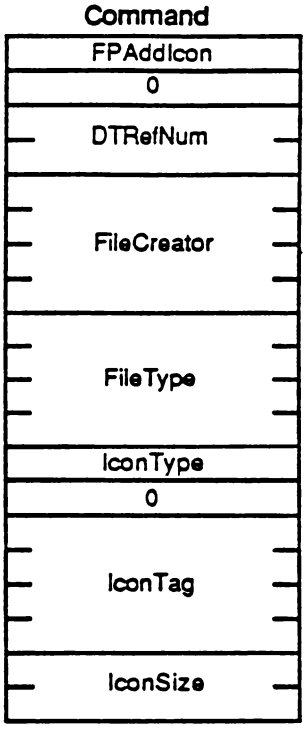
INPUTS:	SRefNum (INT)	session refnum
	DTRefNum (INT)	Desktop Database refnum
	FileCreator (RESTYPE)	file's creator type
	FileType (RESTYPE)	file's type
	IconType (BYTE)	type of icon being added
	IconTag (LONG)	tag information to be stored with the icon
	BitmapSize (INT)	size of the bitmap for this icon
OUTPUTS:	FPError (LONG)	
ERRORS:	ParamErr	unknown session refnum or Desktop Database refnum
	IconTypeError	new icon size is different from existing icon's size
	AccessDenied	user does not have the rights listed below

ALGORITHM: A new icon is added to the Desktop database for the specified FileCreator and FileType. If an icon of the same FileCreator, FileType, and IconType already exists the icon is replaced. If the new icon's size is different from the old icon's size an IconTypeError is returned.

RIGHTS: The user must have previously called FPOpenDT for the corresponding volume. In addition, the volume indicated by DTRefnum must not be marked ReadOnly.

NOTES: The command packet includes all input parameters except for the actual bitmap. The bitmap is sent to the server in a subsequent intermediate exchange of Session Protocol packets.

PACKET FORMAT:


Command
FPAddIcon
0
DTRefNum
FileCreator
FileType
IconType
0
IconTag
IconSize

FPByteRangeLock

This call is used to lock a range of an open fork to ensure exclusive access. Locks prevent all other users from reading or writing any bytes within the range.

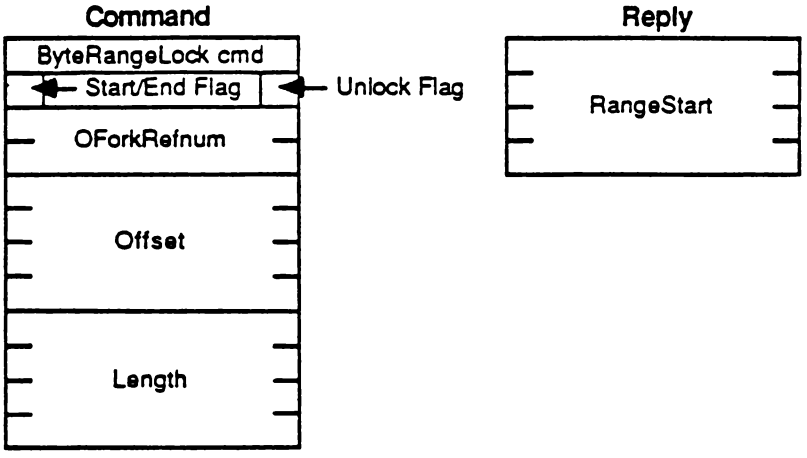
INPUTS:	SRefNum (INT)	session refnum
	OForkRefnum (INT)	open fork refnum
	Offset (LONG)	offset to the first byte of the range to be locked or unlocked (can be negative if Start/End Flag = End)
	Length (LONG)	number of bytes to be locked or unlocked (signed—can't be negative)
	UnlockFlag (BIT)	flag to indicate whether range is to be locked or unlocked: 0 = lock 1 = unlock
	Start/EndFlag (BIT)	flag indicating whether the Offset field is relative to the beginning or end of the fork (valid only when locking) - (all other bits must be zero): 0 = relative to the beginning of the fork 1 = relative to the end of the fork
OUTPUTS:	FPError (LONG)	
	RangeStart (LONG)	number of the first byte of the range just locked (valid only when returned from a successful lock command)
ERRORS:	ParamErr	unknown session refnum or open fork refnum; combination of Start/End flag and offset specified a range starting before the 0th byte
	LockErr	some or all of requested range is locked by another user
	NoMoreLocks	server's maximum lock limit has been reached
	RangeOverlap	user attempted to lock (some or all of) a range that is already locked by the user
	RangeNotLocked	tried to unlock a range that was not locked by the user

ALGORITHM: If no other user holds a lock on any part of the requested range, the server will lock exactly the specified range for this user. A user may hold multiple locks on a given open fork, up to a server-specific limit. Multiple locks may not overlap. A lock range may start and/or extend past the end-of-fork; this does not prevent another user from writing to the fork past the locked range. Specifying an Offset of zero and a Length of $FFFFFFFF will lock the entire fork. All locks held by a user are unlocked when the user closes the fork. Unlocking a range makes it available for reading and writing to other users. A RangeNotLocked error is returned if an attempt was made to unlock a range that was not locked by the user.

If multiple writers are concurrently modifying the fork, they may each have a different notion of the end-of-fork, although the server always knows the correct end-of-fork. It is for this reason that the "lock relative to end-of-fork" feature is provided. The number of the first locked byte is returned, since the end-of-fork may have been different from the user's notion at the time at which the call was made.

RIGHTS: No special access rights are needed to make this call.

PACKET FORMAT:



FPCloseDir

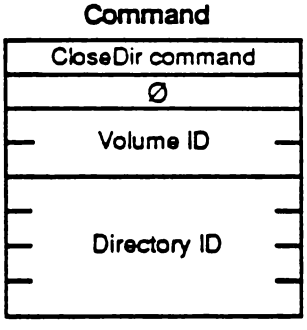
This call is used to close a directory.

INPUTS:	SRefNum (INT)	session refnum
	VolumeID (INT)	volume identifier
	DirID (LONG)	ancestor directory identifier
OUTPUTS:	FPError (LONG)	
ERRORS:	ParamErr	unknown session refnum or volume identifier
	ObjectNotFound	unknown directory identifier

ALGORITHM: The directory identifier is invalidated, and may not be used again.

RIGHTS: The user must have previously called FPOpenVol for this volume and FPOpenDir for this directory.

PACKET FORMAT:

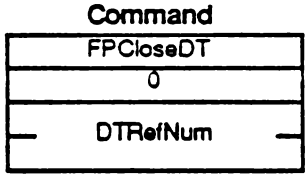


FPCloseDT

This call is used to disassociate a user from the volume's Desktop Database.

INPUTS:	SRefNum (INT) DTRefNum (INT)	session refnum Desktop Database refNum, as returned from FPOpenDT
OUTPUTS:	FPError (LONG)	
ERRORS:	ParamErr	unknown session refnum or Desktop Database refnum
RIGHTS:	The user must have made a successful FPOpenDT call before this call can be made.	

PACKET FORMAT:



FPCloseFork

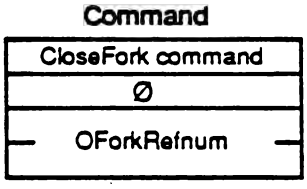
This call is used to close a fork which was opened by FPOpenFork.

INPUTS:	SRefNum (INT) OForkRefnum (INT)	session refnum open fork refnum
OUTPUTS:	FPError (LONG)	
ERRORS:	ParamErr	unknown session refnum or open fork refnum

ALGORITHM: The server flushes and then closes the open fork, invalidating the OForkRefnum. If the fork had been written to, the file's Mod Date will be set to the server's clock at this time.

RIGHTS: No special access rights are needed to make this call.

PACKET FORMAT:

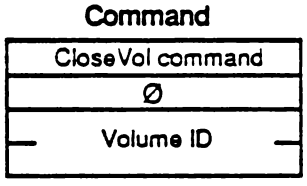


FPCloseVol

This call is used to "unmount" a volume.

INPUTS:	SRefNum (INT) VolumeID (INT)	session refnum volume identifier
OUTPUTS:	FPError (LONG)	
ERRORS:	ParamErr	unknown session refnum or volume identifier
ALGORITHM:	The Volume ID is invalidated. No further calls may be made to access objects on this volume unless another FPOpenVol call is made.	
RIGHTS:	The user must have previously called FPOpenVol for this volume.	

PACKET FORMAT:



FPCopyFile (optional; may not be supported by all servers)

This call is used to copy a file residing on one of the server's volumes to another location on one of the server's volumes. The destination of the copy is specified by providing a VolID, DirID, and Pathname that indicate the copy's new Parent Directory.

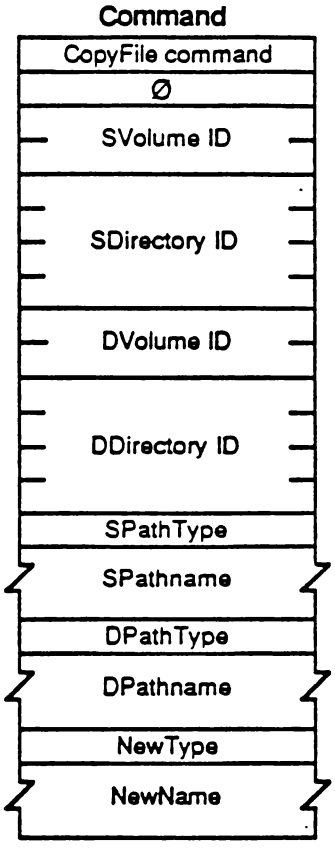
INPUTS:	SRefNum (INT)	session refnum
	SVolumeID (INT)	source volume identifier
	SDirID (LONG)	source ancestor directory identifier
	SPathType (BYTE)	indicates whether SPathname is composed of long names or short names: 1 = all pathname elements are short names 2 = all pathname elements are long names
	SPathname (STR)	pathname of the file to be copied (cannot be null)
	DVolumeID (INT)	destination volume identifier
	DDirID (LONG)	destination ancestor directory identifier
	DPathType (BYTE)	indicates whether DPathname is composed of long names or short names (same values as SPathType)
	DPathname (STR)	pathname to the destination Parent Directory (may be null)
	NewType (BYTE)	indicates whether NewName is a long name or a short name (same values as SPathType)
	NewName (STR)	name to be given to the copy (may be null)
OUTPUTS:	FPError (LONG)	
ERRORS:	ParamErr	unknown session refnum, volume identifier or pathname type; bad pathname or NewName
	ObjectNotFound	the source file does not exist; unknown ancestor directory
	ObjectExists	an object by the name NewName already exists in the destination Parent Directory
	AccessDenied	user does not have the right to read the file or write to the destination
	CallNotSupported	call not supported by this server
	DenyConflict	the file cannot be opened for Read, Deny Write
	DiskFull	no more space on the volume
	ObjectTypeErr	source parameters point to a directory

ALGORITHM: The server will attempt to open the source file for Read, Deny Write. If that fails, a Deny Conflict error will be returned. Otherwise, the file will be copied from source to destination. The original file is not changed or deleted.

The copy is given the name specified in NewName. If NewName is null, the server will attempt to give the copy the same name as the original. The creation of Long and Short names is performed as described in Appendix B. A unique FileNumber is assigned to the file. Its Parent Directory ID will be set to the Dir ID of the destination Parent Directory. All other file parameters remain the same as the source file's parameters. The Mod Date of the destination Parent Directory is set to the server's clock.

RIGHTS: The user must have previously called FPOpenVol for both source and destination volumes. In addition, the user must have Search access rights to all ancestors except the source file's Parent Directory, and Read access right to the source file's Parent directory. Further, the user must have Search access rights to all ancestors except the destination Parent Directory, and Write access right to the destination Parent Directory.

PACKET FORMAT:


Command
CopyFile command
Ø
SVolume ID
SDirectory ID
DVolume ID
DDirectory ID
SPathType
SPathname
DPathType
DPathname
NewType
NewName

FPCreateDir

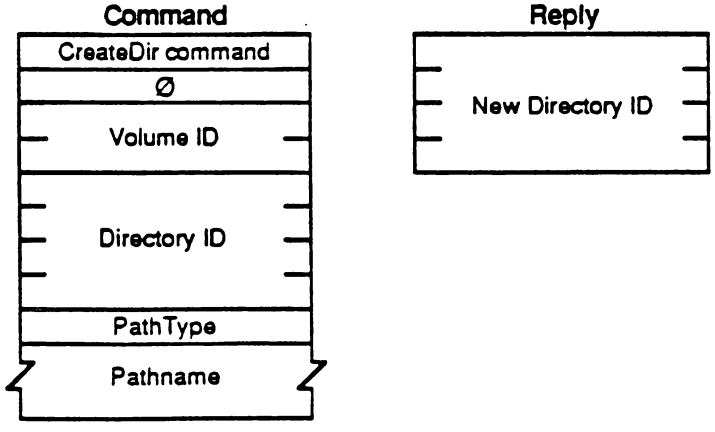
This call is used to create a new directory.

INPUTS:	SRefNum (INT)	session refnum
	VolumeID (INT)	volume identifier
	DirID (LONG)	ancestor directory identifier
	PathType (BYTE)	indicates whether Pathname is composed of long names or short names: 1 = all pathname elements are short names 2 = all pathname elements are long names
	Pathname (STR)	pathname to desired directory (cannot be null)
OUTPUTS:	FPError (LONG)	
	NewDirID (LONG)	identifier of new directory
ERRORS:	ParamErr	unknown session refnum, volume identifier, or pathname type; null or bad pathname
	ObjectNotFound	unknown ancestor directory
	ObjectExists	an object already exists by that name
	AccessDenied	user does not have the rights listed below
	FlatVol	the volume is flat and does not support directories
	DiskFull	no more space on the volume

ALGORITHM: If the volume is not flat, a new empty directory is created with the name as specified in Pathname. A unique NewDirID is assigned to the directory. Its OwnerID is set to the UserID of the user making the call, and its GroupID is set to the ID of the user's Primary Group. Access rights are initially set to Read, Write, and Search for the Owner, no rights for the Group or World. Finder Info is zeroed. Create Date and Mod Date are set to the server's clock. Backup Date is set to $80000000, signifying that this directory has never been backed up. The creation of Long and Short names is performed as described in Appendix B. All attributes are initially cleared. The Mod Date of the Parent Directory is set to the server's clock.

RIGHTS: The user must have previously called FPOpenVol for this volume. In addition, the user must have Search or Write access rights to all ancestors except this directory's Parent Directory, as well as Write access right to the Parent Directory.

PACKET FORMAT:



FPCreateFile

This call is used to create a file.

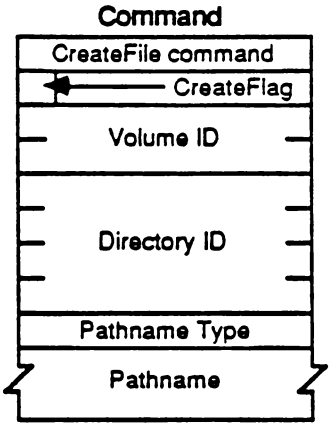
INPUTS:	SRefNum (INT)	session refnum
	VolumeID (INT)	volume identifier
	DirID (LONG)	ancestor directory identifier
	CreateFlag (BIT)	a flag that specifies hard or soft create (all other bits will be zero): 0 = soft create 1 = hard create
	PathType (BYTE)	indicates whether Pathname is composed of long names or short names: 1 = all pathname elements are short names 2 = all pathname elements are long names
	Pathname (STR)	pathname including name of new file (cannot be null)
OUTPUTS:	FPError (LONG)	
ERRORS:	ParamErr	unknown session refnum, volume identifier, or pathname type; null or bad pathname
	ObjectNotFound	unknown ancestor directory
	ObjectExists	soft create: a file by that name already exists
	ObjectTypeErr	a directory by that name already exists
	AccessDenied	user does not have the rights listed below
	FileBusy	hard create: the file already exists and is open
	DiskFull	no more space on the volume

ALGORITHM: In a soft create, if the object does not already exist, a new file is created with the name as specified in Pathname. A unique FileNumber is assigned to the file. Finder Info is zeroed. Create Date and Mod Date are set to the server's clock. Backup Date is set to $80000000, signifying that this file has never been backed up. The creation of Long and Short names is performed as described in Appendix B. The lengths of both forks are set to zero. The Mod Date of the file's Parent Directory is set to the server's clock. All file attributes are initially cleared. The Mod Date of the Parent Directory is set to the server's clock.

In a hard create, if the file already exists, it is essentially deleted and then recreated. All file parameters (including the Create Date) are reinitialized as described above.

RIGHTS: The user must have previously called FPOpenVol for this volume. For a soft create, the user must have Search or Write access rights to all ancestors except this file's Parent Directory, as well as Write access right to the Parent Directory. For a hard create, the user must have Search access to all ancestors except the Parent, as well as Read and Write access to the Parent.

PACKET FORMAT:



FPDelete

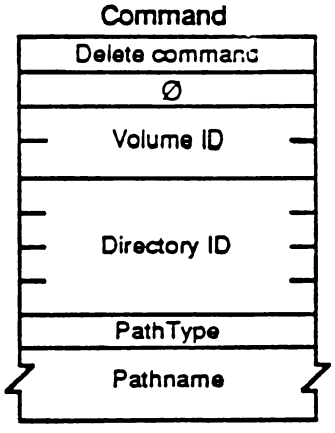
This call is used to delete either a directory or file.

INPUTS:	SRefNum (INT)	session refnum
	VolumeID (INT)	volume identifier
	DirID (LONG)	ancestor directory identifier
	PathType (BYTE)	indicates whether Pathname is composed of long names or short names: 1 = all pathname elements are short names 2 = all pathname elements are long names
	Pathname (STR)	pathname of file or directory to be deleted (may be null if a directory is to be deleted)
OUTPUTS:	FPError (LONG)	
ERRORS:	ParamErr	unknown session refnum, volume identifier, or pathname type; bad pathname
	ObjectNotFound	input parameters do not point to an existing file or directory
	DirNotEmpty	the directory is not empty
	FileBusy	the file is open
	AccessDenied	user does not have the rights listed below

ALGORITHM: If the object to be deleted is a directory, the server checks to see if it contains any offspring; a DirNotEmpty error is returned if so. If a file is to be deleted, it must not be currently open by any user (else a FileBusy error is returned). The Mod Date of the object's Parent Directory is set to the server's clock.

RIGHTS: The user must have previously called FPOpenVol for the volume. In addition, the user must have Search access rights to all ancestors except the object's Parent Directory, as well as Write access right to the Parent Directory. If a directory is being deleted, the user must also have Search access to the Parent; for a file, the user must also have Read access to the Parent Directory.

PACKET FORMAT:



FPEnumerate

This call is used to enumerate the contents of a directory. The reply is composed of a number of file and/or directory parameter structures.

INPUTS:

SRefNum (INT)	session refnum
VolumeID (INT)	volume identifier
DirID (LONG)	ancestor directory identifier
FileBitmap (INT)	bitmap describing which parameters are to be returned if the enumerated object is a file (the corresponding bit should be set). This field is the same as that in the FPGetFileDirParms call, and may be null.
DirBitmap (INT)	bitmap describing which parameters are to be returned if the enumerated object is a directory (the corresponding bit should be set). This field is the same as that in the FPGetFileDirParms call, and may be null.
ReqCount (INT)	maximum number of structures to be returned
StartIndex (INT)	directory offspring index, described below
MaxReplySize (INT)	maximum size of reply buffer
PathType (BYTE)	indicates whether Pathname is composed of long names or short names: 1 = all pathname elements are short names 2 = all pathname elements are long names
Pathname (STR)	pathname to desired directory

OUTPUTS:

FPError (LONG)	
FileBitmap (INT)	copy of input parameter
DirBitmap (INT)	copy of input parameter
ActCount (INT)	actual number of structures returned (zero if error)

ActCount number of file/directory structures of the form:

StructLength (BYTE)	unsigned length of this structure, including these two 'header' bytes, and rounded up to the nearest even number
File/DirFlag (BIT)	flag indicates whether structure describes a file or directory: 0 = file; 1 = directory. All other bits must be zero.
File or Directory Parameters, packed in Bitmap order, with a trailing null BYTE if necessary to make the length of the entire structure even	

ERRORS:

ParamErr	unknown session refnum, volume identifier, directory identifier, or pathname type; bad pathname; MaxReplySize is too small to hold even a single entry
DirNotFound	input parameters do not point to an existing directory
BitmapErr	an attempt was made to retrieve a parameter which cannot be retrieved with this call; both bitmaps are empty
AccessDenied	user does not have the rights listed below
ObjectNotFound	no more offspring to enumerate
ObjectTypeErr	input parameters pointed to a file

ALGORITHM: The server does an enumeration of the directory as specified with the input parameters: if the FileBitmap is empty, only directory offspring will be enumerated, and the StartIndex may range from 1 to the total number of directory offspring. Likewise, if the DirBitmap is empty, only file offspring will be enumerated, and the StartIndex may range from 1 to the total number of file offspring. If both Bitmaps are non-empty, the StartIndex may range from 1 to the total number of offspring, and structures for both files and directories will be returned. These structures are not returned in any particular order.

This call will complete when ReqCount structures have been inserted into the Reply packet (no partial structures will be returned), or when the Reply packet is full, or when there are no more offspring to enumerate.

The server retrieves the specified parameters for each enumerated offspring and packs them, in Bitmap order, in structures in the reply packet along with copies of the input Bitmaps inserted before the structures. In order to keep all variable-length parameters at the end of each structure (even if more parameters are later added), all such parameters like the Long Name and Short Name fields will be represented in the Bitmap order as fixed-length offsets (INTs) from the start of the parameters in each structure (not the start of the bitmap and not the start of the 'header' bytes) to the start of the variable-length fields. Each structure may be null-padded to make its length even.

A BitmapErr will be returned if an attempt is made to retrieve the DirectoryID parameter for a directory on a Variable-DirID volume.

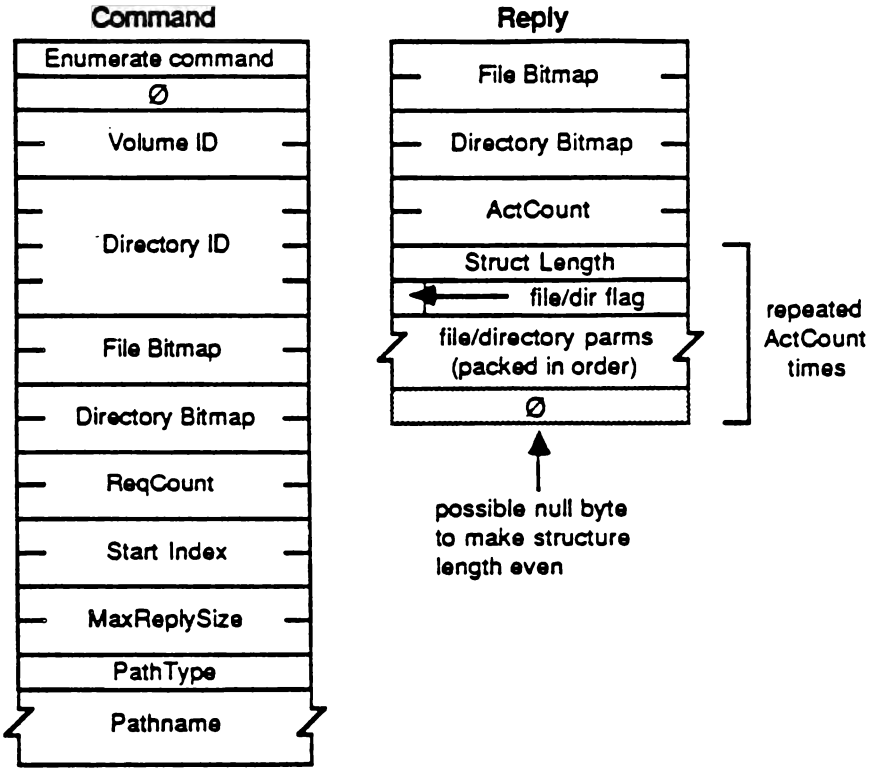
If NoErr is returned, then all the structures returned in the Reply packet will be valid. If any error occurs, there will be no valid structures in the Reply packet.

RIGHTS: The user must have previously called FPOpenVol for this volume. In addition, the user must have Search access rights to all ancestors except this directory; Search access right to this directory is needed to be able to enumerate directory offspring. Read access right is needed to this Directory in order to be able to enumerate file offspring.

NOTES: Since enumerating a large directory may take several calls, it is possible (since other users may be adding to or deleting from the directory) for the enumeration to miss offspring or return duplicate offspring. To be safe, keep enumerating until an ObjectNotFound error is returned, and filter out duplicate entries.

A given offspring is not guaranteed to occupy the same index number in the Parent Directory from one enumeration to the next.

PACKET FORMAT:



FPFlush

This call is used to flush to disk any data relating to the specified volume that has been modified by the user.

INPUTS:	SRefNum (INT) VolumeID (INT)	session refnum volume identifier
OUTPUTS:	FPError (LONG)	
ERRORS:	ParamErr	unknown session refnum or volume identifier

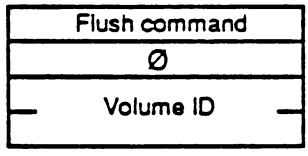
ALGORITHM: The server attempts to flush to disk as much changed information as possible. This includes (a) flushing all forks opened by the user, (b) flushing catalog information changed by the user, and (c) flushing any updated volume-level information. Since it may be difficult or impossible for all servers to guarantee that this can all be done, the above list is meant as a suggestion. Users should not rely on any or all of the above actions to actually be done.

The volume's Mod Date may change as a result of this call, but users should not rely on it since updating of the date is implementation-dependent. If no volume information was changed since the last FPFlush call, the date may or may not change.

RIGHTS: The user must have previously called FPOpenVol for this volume.

PACKET FORMAT:

Command



FPFlushFork

Any FPWrites made to a particular file fork may be buffered by the server in order to optimize disk accesses. Within the constraints of performance, the server will try to flush (commit to disk) each file as soon as possible, yet clients can force the server to write to the disk any data buffered from previous FPWrites by issuing this call.

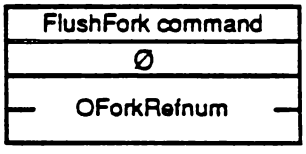
INPUTS:	SRefNum (INT) OForkRefnum (INT)	session refnum open fork refnum
OUTPUTS:	FPError (LONG)	
ERRORS:	ParamErr	unknown session refnum or open fork refnum

ALGORITHM: The server will commit to disk all cached writes to the fork, and set the file's Mod Date to the server's clock if the fork was written to.

RIGHTS: No special access rights are needed to make this call.

PACKET FORMAT:

Command



FPGetAPPL

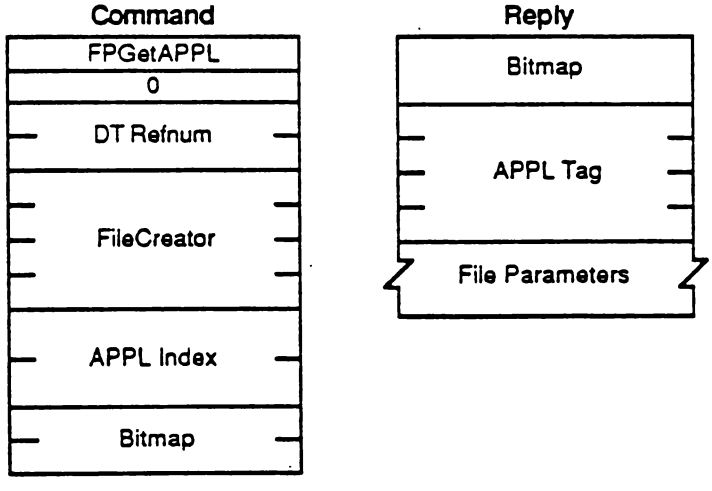
This call is used to retrieve information about a particular application from the Desktop Database.

INPUTS::	SRefNum (INT)	session refnum
	DTRefnum (INT)	Desktop Database refnum
	FileCreator (RESTYPE)	creator type of the application to be returned
	APPL Index (INT)	index of the APPL entry to be retrieved
	Bitmap (INT)	bitmap indicating the parameters of the application file to be returned. This field is the same as the FileBitmap in the FPGetFileDirParms call.
OUTPUTS:	FPError (LONG)	
	APPLTag (LONG)	tag information associated with the APPL entry
ERRORS:	ParamErr	unknown session refnum or Desktop Database refnum
	ObjectNotFound	no files in the Desktop Database match input parameters
	AccessDenied	user does not have the rights listed below

ALGORITHM: The entries under the specified FileCreator are examined, and the n^{th} entry, as indicated by the APPL index, is returned. Entries for applications which are not accessible by the user are not returned.

RIGHTS: The user must have previously called FPOpenDT for the corresponding volume, and must have Search access to all ancestors except the Parent and Read access to the Parent of the application whose information will be returned.

PACKET FORMAT:



FPGetComment

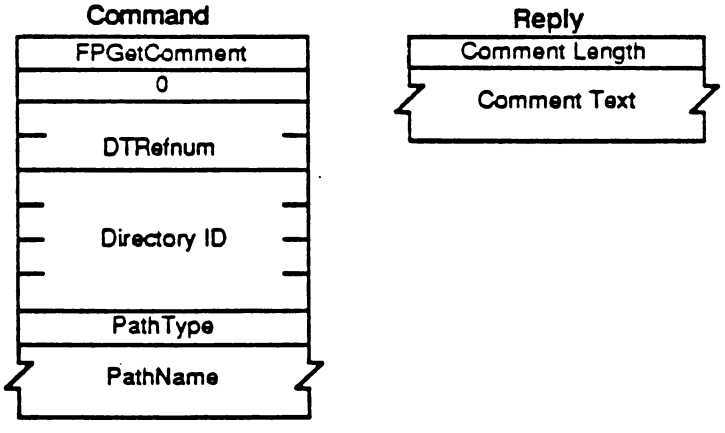
This call is used to retrieve a comment associated with a specified file or directory from the Desktop Database.

INPUTS:	SRefNum (INT)	session refnum
	DTRefnum (INT)	Desktop Database Refnum
	DirID (LONG)	directory identifier
	PathType (BYTE)	indicates whether Pathname is composed of long names or short names: 1 = all pathname elements are short names 2 = all pathname elements are long names
	Pathname (STR)	pathname to desired object
OUTPUTS:	FPError (LONG)	
	CmtLength (BYTE)	length of the comment data
	CmtText (BYTES)	comment text
ERRORS:	ParamErr	unknown session refnum or Desktop Database refnum
	ObjectNotFound	input parameters do not point to an existing file or dir
	AccessDenied	user does not have the rights listed below
	ItemNotFound	no comment was found in the Desktop Database

ALGORITHM: The comment for the specified file is located in the Desktop Database and returned to the caller.

RIGHTS: The user must previously have called FPOpenDT for the corresponding volume. In addition, the file or directory must be present before this call is issued. If the comment is associated with a directory, the user must have Search access to all ancestors including the Parent Directory. If the comment is associated with a file, the user must have Search access to all ancestors except the Parent Directory, and Read access to the Parent Directory.

PACKET FORMAT:



FPGetFileDirParms

This call is used to retrieve parameters for an object that may be a file or a directory.

INPUTS:	SRefNum (INT)	session refnum
	VolumeID (INT)	volume identifier
	DirID (LONG)	ancestor directory identifier
	FileBitmap (INT)	bitmap describing which parameters are to be returned if the object is a file (the corresponding bit should be set): 0 (LSB) Attributes (INT), consisting of the following flags: 0 Invisible 1 Multi-User 3 DAlreadyOpen 4 RAlreadyOpen 5 ReadOnly 15 Set/Clear (used in FPSetFileDirParms) 1 Parent Directory ID (LONG) 2 Create Date (LONG) 3 Mod Date (LONG) 4 Backup Date (LONG) 5 Finder Info (32 BYTEs) 6 Long Name (INT) 7 Short Name (INT) 8 File Number (LONG) 9 Data Fork Length (LONG) 10 Resource Fork Length (LONG)
	DirBitmap (INT)	bitmap describing which parameters are to be returned if the object is a directory (the corresponding bit should be set): 0 (LSB) Attributes (INT), consisting of the following flag: 0 Invisible 1 Parent Directory ID (LONG) 2 Create Date (LONG) 3 Mod Date (LONG) 4 Backup Date (LONG) 5 Finder Info (32 BYTEs) 6 Long Name (INT) 7 Short Name (INT) 8 Directory ID (LONG) 9 Number of Offspring (INT) 10 Owner ID (LONG) 11 Group ID (LONG) 12 Access Rights (LONG), composed of the access privileges for Owner, Group, and World, and a User Rights Summary BYTE
	PathType (BYTE)	indicates whether Pathname is composed of long names or short names: 1 = all pathname elements are short names 2 = all pathname elements are long names
	Pathname (STR)	pathname to desired object
OUTPUTS:	FPError (LONG)	
	FileBitmap (INT)	copy of input parameter
	DirBitmap (INT)	copy of input parameter
	ObjectFlag (BIT)	one-bit flag that indicates whether object is a file or a directory: 0 = file; 1 = directory. All other bits should be zero.

Parameters

ERRORS:	ParamErr	unknown session refnum, volume identifier, or pathname type; bad pathname
	ObjectNotFound	input parameters do not point to an existing file or dir
	BitmapErr	an attempt was made to retrieve a parameter which cannot be obtained with this call
	AccessDenied	user does not have the rights listed below

ALGORITHM: The server retrieves the specified parameters for the object and packs them, in the order specified by the appropriate Bitmap, in the reply packet along with a flag indicating the type of object and a copy of the Bitmaps inserted before the parameters. In order to keep all variable-length parameters at the end of the packet (even if more parameters are later added), all such parameters like the Long Name and Short Name fields will be represented in the Bitmap order as fixed-length offsets (INTs) from the start of the parameters (not the start of the bitmap) to the variable-length fields. The actual variable-length fields are then packed after all fixed-length fields.

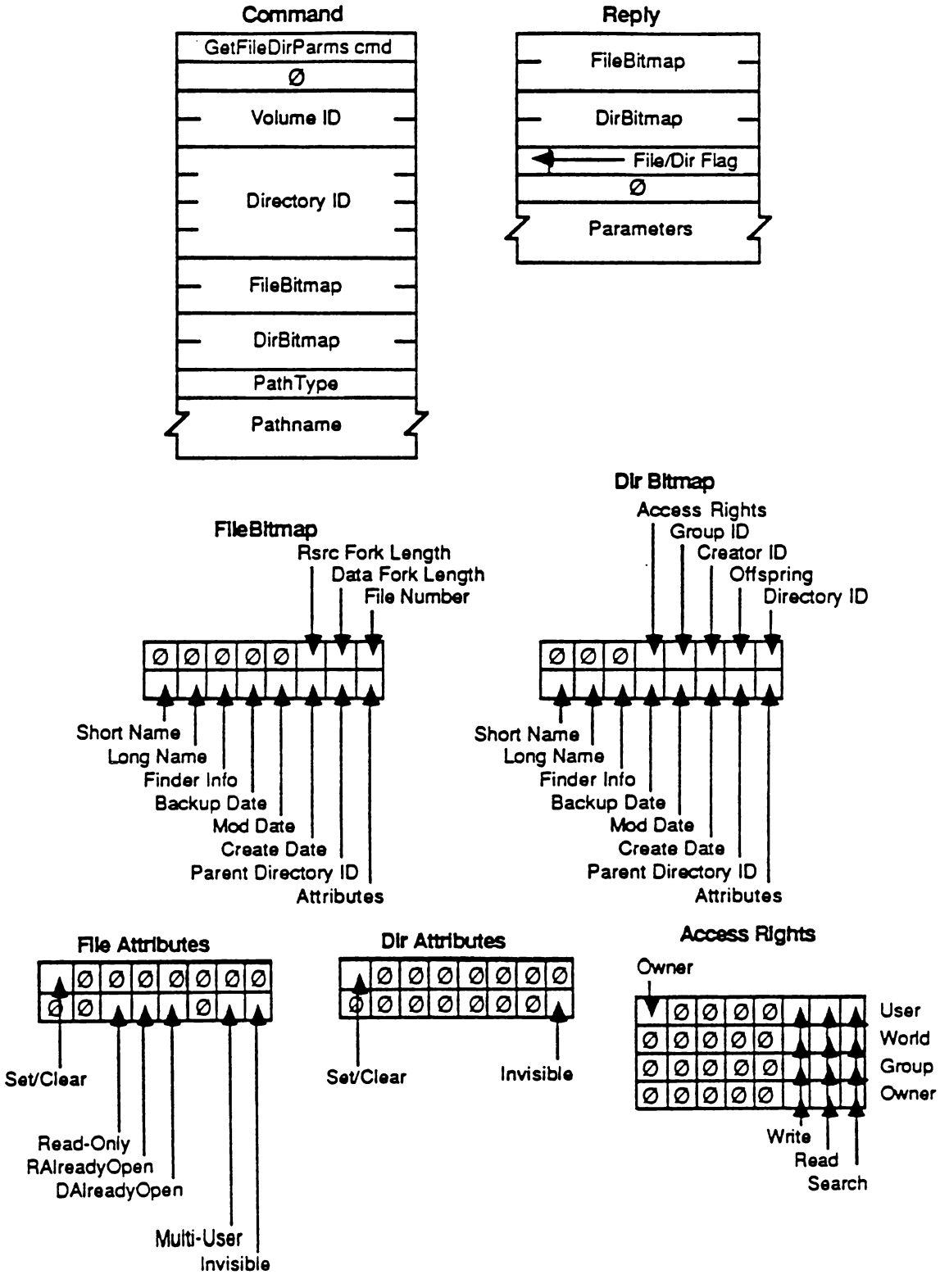
If the object exists but both bitmaps are null, no error will be returned. The File Bitmap, Dir Bitmap, and File/Dir Flag will be returned with no other parameters.

If the Access Rights for a directory are requested, the server will return a LONG containing the Read, Write, and Search access priviliges corresponding to Owner, Group, and World. In addition, the upper byte of the Access Rights LONG is the User Rights Summary byte, indicating what privileges the user has to this directory, and whether or not the user is the owner of the directory. This Owner Bit is also set if the directory is owned by <any user>.

RIGHTS: The user must have previously called FPOpenVol for this volume. In addition, the user must have Search access rights to all ancestors except this object's Parent Directory. If the object is a directory, the user also needs Search access to the Parent Directory; else if the object is a file, the user needs Read access to the Parent Directory.

NOTES: Most Attributes are actually stored in corresponding flags within the FinderInfo field.

PACKET FORMAT:



FPGetForkParms

This call is used to retrieve parameters for a file associated with a particular open fork.

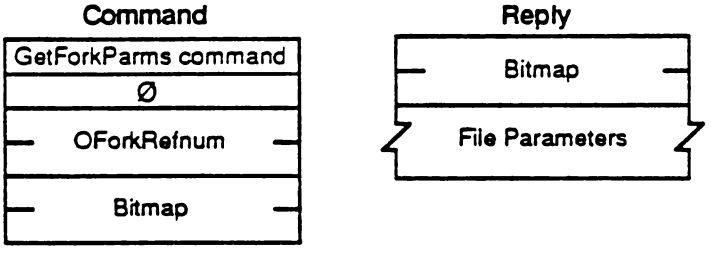
INPUTS:	SRefNum (INT)	session refnum
	OForkRefnum (INT)	open fork refnum
	Bitmap (INT)	bitmap describing which parameters are to be retrieved (the corresponding bit should be set). This field is the same as that in the FPGetFileDirParms call.
OUTPUTS:	FPError (LONG)	
	Bitmap (INT)	copy of the input parameter
	File Parameters	
ERRORS:	ParamErr	unknown session refnum or open fork refnum
	BitmapErr	an attempt was made to retrieve a parameter which cannot be obtained with this call; null bitmap
	AccessDenied	fork was not opened for Read

ALGORITHM: The server retrieves the specified parameters for the file and packs them, in Bitmap order, in the reply packet. In order to keep all variable-length parameters at the end of the packet (even if more parameters are later added), all such parameters will be represented in the Bitmap order as fixed-length offsets (INTs) from the start of the parameters to the start of the variable-length fields. The actual variable-length fields are then packed after all fixed-length fields.

In AFP Version 1.1, the length of the fork indicated by the OForkRefnum may be retrieved, but a BitmapErr will be returned if an attempt is made to retrieve the length of the other fork comprising the file.

RIGHTS: The fork must have been opened for Read.

PACKET FORMAT:



FPGetIcon

This call is used to retrieve an icon from the Desktop database from a FileCreator/FileType specification.

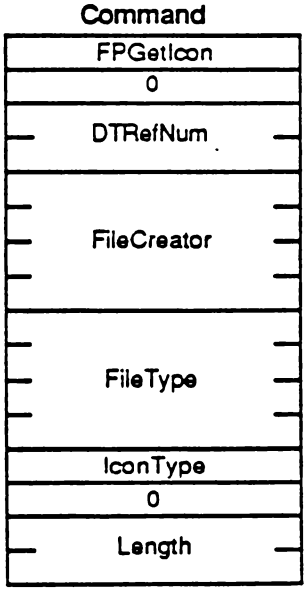
INPUTS:	SRefNum (INT)	session refnum
	DTRefnum (INT)	Desktop Database refnum
	FileCreator (RESTYPE)	File's Creator type
	FileType (RESTYPE)	File's type
	IconType (BYTE)	Preferred icon type
	Length (INT)	the number of bytes reserved for icon bitmap
OUTPUTS:	FPError (LONG)	
	Icon Bitmap (BYTES)	The actual bitmap for the icon
ERRORS:	ParamErr	unknown session refnum or Desktop Database refnum
	ItemNotFound	no icon corresponding to the input specification was found in the Desktop Database

ALGORITHM: The bitmap for the specified icon is looked up in the Desktop Database given its FileCreator, FileType, and IconType and returned to the caller if found. If no matching icon is found, an ItemNotFound error will be returned.

Note that an input length argument of zero is acceptable to test for the presence or absence of a particular icon. The size of the bitmap returned is the minimum of the requested length and the actual size of the icon.

RIGHTS: The user must have previously called FPOpenDT for the corresponding volume.

PACKET FORMAT:



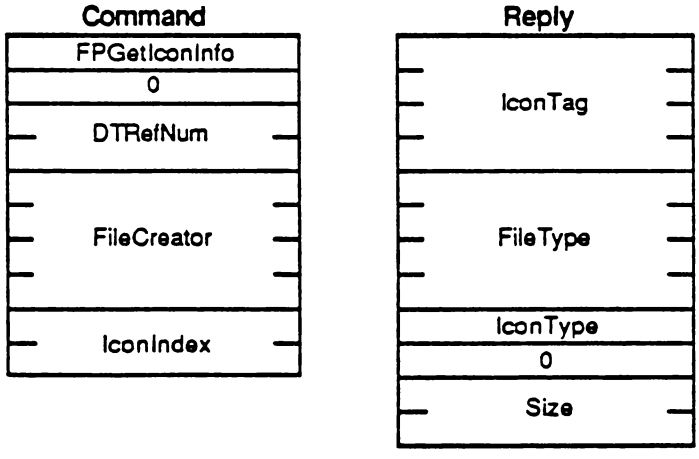
FPGetIconInfo

INPUTS:	SRefNum (INT)	session refnum
	DTRefnum (INT)	Desktop Database refnum
	FileCreator (RESTYPE)	File's Creator type
	IconIndex (INT)	Index of requested icon
OUTPUTS:	FPError (LONG)	
	IconTag (LONG)	Tag information associated with the requested icon
	FileType (RESTYPE)	The file type of the requested icon
	IconType (BYTE)	The type of the requested icon
	Size (INT)	The size of the icon bitmap
ERRORS:	ParamErr	unknown session refnum or Desktop Database refnum
	ItemNotFound	no icon corresponding to the input specification was found in the Desktop Database

ALGORITHM: The Icon Index argument is used to determine the n^{th} icon for the given Creator type to be returned. If the icon index is greater than the number of icons in the Desktop Database for the specified Creator type, ItemNotFound is returned.

RIGHTS: The user must have previously called FPOpenDT for the corresponding volume.

PACKET FORMAT:



FPGetSrvrInfo

This call is used to obtain a block of descriptive information from the server, without requiring a session to be opened.

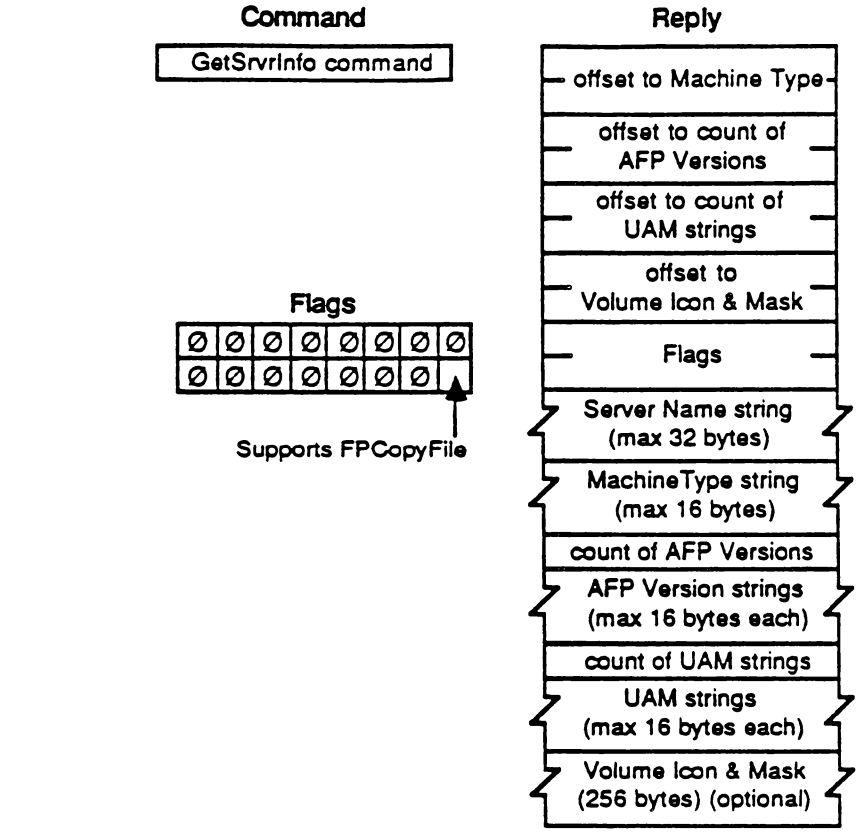
INPUTS:	SAddr (EntityAddr)	network-dependent internet address of the file server
OUTPUTS:	FPError (LONG)	
	Flags (INT)	Flags, consisting of: Bit 0 SupportsCopyFile — set if server supports the FPCopyFile call
	Server Name (STR)	the name of the server
	Machine Type (STR)	string describing the server's hardware and/or OS
	AFP Versions (STRs)	versions of AFP that the server can speak
	UAM strings (STRs)	User Authentication Methods supported by the server
	Volume Icon and Mask (256 BYTEs)	
ERRORS:	NoServer	server not responding

ALGORITHM: The info block is returned from the server. The AFP Versions and UAM strings are formatted as a one-BYTE count followed by that number of strings packed back-to-back without padding. To facilitate access to all the fields of the reply packet, the packet is formatted as shown below: the packet data begins with INT offsets to the Machine Type, AFP Versions, UAM strings, and Volume Icon and Mask. These offsets are measured relative to the start of the reply packet data. The Volume Icon and Mask field is optional; if it is not included, the Offset to Volume Icon and Mask will be zero.

RIGHTS: No special access rights are needed to make this call.

NOTES: The server may pack fields in the Reply block in any order; each field should be accessed only via the offsets (make no assumptions about how the fields are packed relative to one another). The exception to this is that the Server Name string (for which there is no offset) begins immediately after the Flags field.

PACKET FORMAT:



FPGetSrvrParms

This call is used to retrieve server-level parameters.

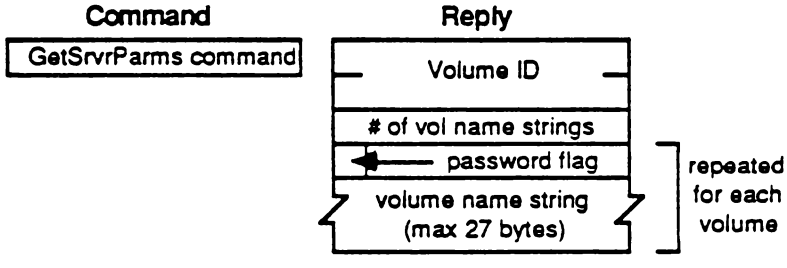
INPUTS:	SRefNum (INT)	session refnum
OUTPUTS:	FPError (LONG)	
	ServerTime (LONG)	current date-time on this server's clock
	NumVols (BYTE)	number of volumes managed by the server
	HasPassword (BIT)	flag indicating whether or not this volume is password-protected: 0 = not protected; 1 = has password (all other bits must be zero)
	VolNames (STRs)	character string names of each volume (maximum 27 bytes)
ERRORS:	ParamErr	unknown session refnum

ALGORITHM: The volume name strings and password flags are packed together without padding in the reply.

RIGHTS: No special access rights are needed to make this call.

NOTES: This call should be implemented on a server using the ASP GetStatus mechanism.

PACKET FORMAT:



FPGetVolParms

This call is used to retrieve parameters for a particular volume. The volume is specified by its Volume ID as returned from the FPOpenVol call.

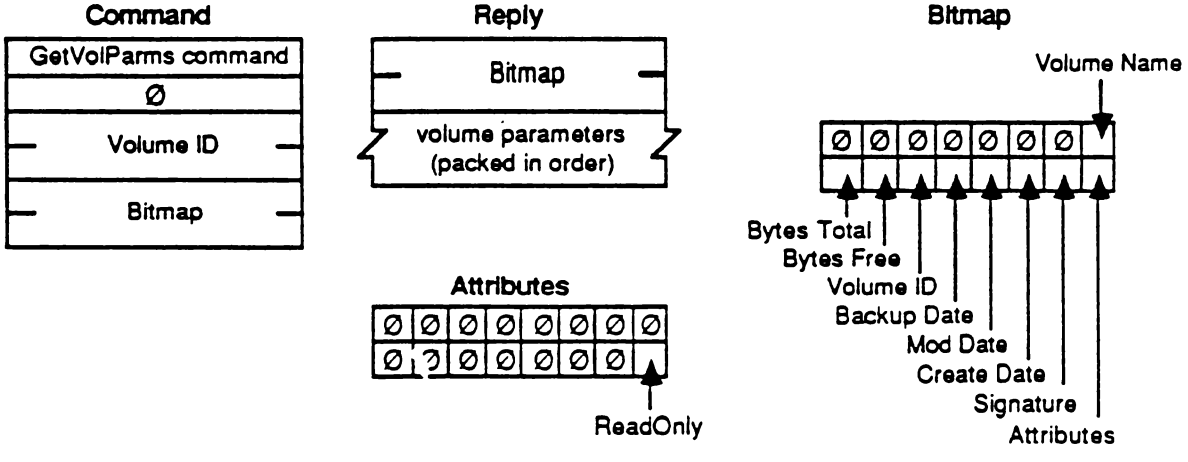
INPUTS:	SRefNum (INT)	session refnum
	VolumeID (INT)	volume identifier
	Bitmap (INT)	bitmap describing which parameters are to be returned (the corresponding bit should be set) (cannot be null):
		0 (LSB) Attributes (INT), consisting of the following flag:
		0 ReadOnly
		1 Signature (INT)
		2 Create Date (LONG)
		3 Mod Date (LONG)
		4 Backup Date (LONG)
		5 Volume ID (INT)
		6 Bytes Free (LONG) unsigned
		7 Bytes Total (LONG) unsigned
		8 Volume Name (INT)
OUTPUTS:	FPError (LONG)	
	Bitmap (INT)	copy of input parameter
	Volume Parameters	
ERRORS:	ParamErr	unknown session refnum or volume identifier
	BitmapErr	an attempt was made to retrieve a parameter which cannot be obtained with this call; null bitmap

ALGORITHM: The server retrieves the specified parameters for the volume and packs them, in Bitmap order, in the reply packet along with a copy of the Bitmap inserted before the parameters. In order to keep all variable-length parameters at the end of the packet (even if more parameters are later added), all such parameters like the Volume Name field will be represented in the Bitmap order by fixed-length offsets (INTs) from the start of the parameters (not the start of the bitmap) to the start of the variable-length fields. The actual variable-length fields are then packed after all fixed-length fields.

RIGHTS: The user must have previously called FPOpenVol for this volume.

NOTES: The ReadOnly attribute is intended to be set via some administrative function, not through this protocol.

PACKET FORMAT:



FPLogin

This call is used to establish a session with a server. A protocol version is agreed upon and the user is authenticated.

INPUTS:	SAddr (EntityAddr)	network-dependent internet address of the file server
	AFPVersion (STR)	a string indicating which AFP Version is to be used
	UAM (STR)	a string indicating which User Authentication Method is to be used to authenticate the user
	UserAuthInfo (BUF)	information required to authenticate the user, dependent on the method used (may be null)
OUTPUTS:	FPError (LONG)	
	SRefNum (INT)	session refnum to be used to refer to this session in all subsequent calls (valid if NoErr or AuthContinue error returned)
	IDNumber (INT)	returned in certain UserAuthenticationMethods to be presented in the next FPLoginCont call (only valid if AuthContinue error returned)
	UserAuthInfo (BUF)	returned in certain UserAuthenticationMethods (only valid if AuthContinue error returned)
ERRORS:	NoServer	server not responding
	BadVersNum	server cannot speak the specified AFP version
	BadUAM	unknown UserAuthenticationMethod
	ParamErr	unknown User
	UserNotAuth	UserAuthenticationMethod failed
	AuthContinue	authentication not yet complete
	ServerGoingDown	the server is in the process of shutting down
	MiscErr	user is already authenticated

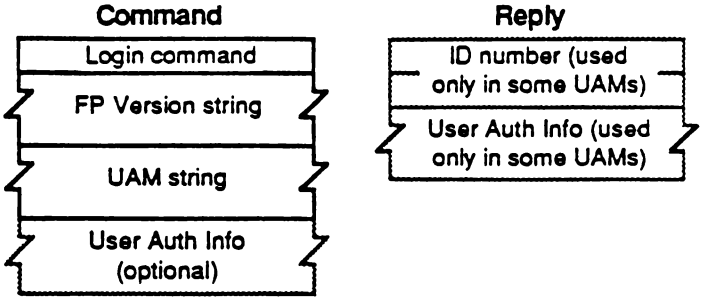
ALGORITHM: The AFP Version string, indicating which AFP Version to use, and the User Authentication Method string, indicating which UAM is to be used in authenticating the user, are sent to the server. These are packed into the command packet with no padding. Depending on which method is used, the command packet sent to the server may contain additional information like user name and password. This extra information is passed to AFP as UserAuthInfo. If the server knows how to execute that UserAuthenticationMethod, it will do so and return a UserNotAuthenticated error if that method fails. Depending on which UAM is used, there may or may not be a null byte padded between the UAM and UserAuthInfo fields. See Appendix A for more details.

Note that some UserAuthenticationMethods may return some Reply data to this call. "No User Authent" and "Cleartxt passwrd" do not. Some methods like "Randnum exchange" will return UserAuthInfo data and will require an additional exchange of packets as well. In such cases, an AuthContinue error will be returned to this call, indicating that subsequent FPLoginCont calls are needed. (See Appendix A for more details.)

NOTES: If any error (other than AuthContinue) is returned, the session will not be opened.

RIGHTS: No special access rights are needed to make this call.

PACKET FORMAT:



FPLoginCont

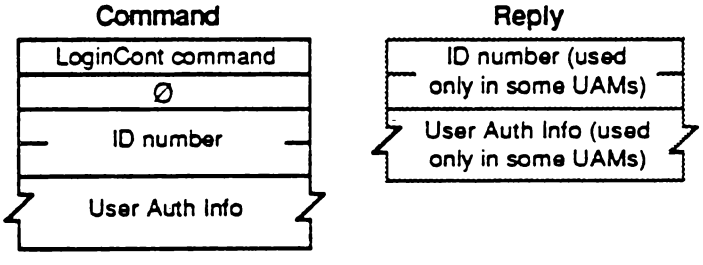
This call is used to continue the Login and authentication process with a server.

INPUTS:	SRefNum (INT)	session refnum
	IDNumber (INT)	number returned from the previous FPLogin or FPLoginCont call
	UserAuthInfo (BUF)	information required to authenticate the user, dependent on the method used
OUTPUTS:	FPError (LONG)	
	IDNumber (INT)	returned in certain UserAuthenticationMethods to be presented in the next FPLoginCont call (only valid if AuthContinue error returned)
	UserAuthInfo (BUF)	returned in certain UserAuthenticationMethods (only valid if AuthContinue error returned)
ERRORS:	NoServer	server not responding
	UserNotAuth	UserAuthenticationMethod failed
	AuthContinue	authorization not yet complete

ALGORITHM: The ID number and UserAuthInfo are sent to the server, which uses them to execute the next step in the authentication method. If an additional exchange of packets is required, an AuthContinue error will be returned. Otherwise, either NoErr (meaning the user has been authenticated) or UserNotAuth (meaning the authentication method has failed) will be returned. If NoErr, a valid SRefNum will be returned for use in subsequent calls. If UserNotAuth, the session is closed by the server and the SRefnum is invalidated.

RIGHTS: No special access rights are needed to make this call.

PACKET FORMAT:



FPLogout

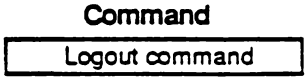
This call is used to terminate a session with a server

INPUTS:	SRefNum (INT)	session refnum
OUTPUTS:	FPError (LONG)	
ERRORS:	ParamErr	unknown session refnum

ALGORITHM: The server flushes and closes any forks opened by this session, frees up all session-related resources and invalidates the session refnum.

RIGHTS: No special access rights are needed to make this call.

PACKET FORMAT:



FPMapID

This call is used to map a User ID to a User Name, or a Group ID to a Group Name.

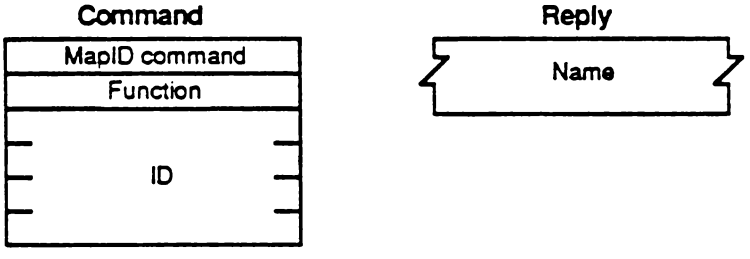
INPUTS:	SRefNum (INT)	session refnum
	Function (BYTE)	function code: 1 = map User ID to User Name 2 = map Group ID to Group Name
	ID (LONG)	item to be mapped, either User ID or Group ID
OUTPUTS:	FPError (LONG)	
	Name (STR)	name corresponding to input ID
ERRORS:	ParamErr	unknown session refnum or function code; no ID was passed in command packet
	ItemNotFound	ID not recognized

ALGORITHM: The server attempts to find the Creator Name or Group Name corresponding to the specified Creator ID or Group ID. An ItemNotFound error is returned if the ID does not exist in the server's list of valid User or Group IDs.

RIGHTS: No special access rights are needed to make this call.

NOTES: A User ID or Group ID of zero will map to the null string.

PACKET FORMAT:



FPMapName

This call is used to map a User Name to a User ID, or a Group Name to a Group ID.

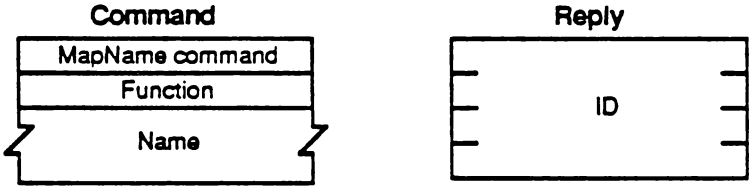
INPUTS:	SRefNum (INT)	session refnum
	Function (BYTE)	function code: 3 = map User Name to User ID 4 = map Group Name to Group ID
	Name (STR)	item to be mapped, either User Name or Group Name
OUTPUTS:	FPError (LONG)	
	ID (LONG)	ID corresponding to input Name
ERRORS:	ParamErr	unknown session refnum or function code
	ItemNotFound	name not recognized

ALGORITHM: The server attempts to find the User ID or Group ID corresponding to the specified User Name or Group Name. An ItemNotFound error is returned if the name does not exist in the server's list of valid User or Group names.

RIGHTS: No special access rights are needed to make this call.

NOTES: A null User or Group Name will map to an ID of zero.

PACKET FORMAT:



FPMove

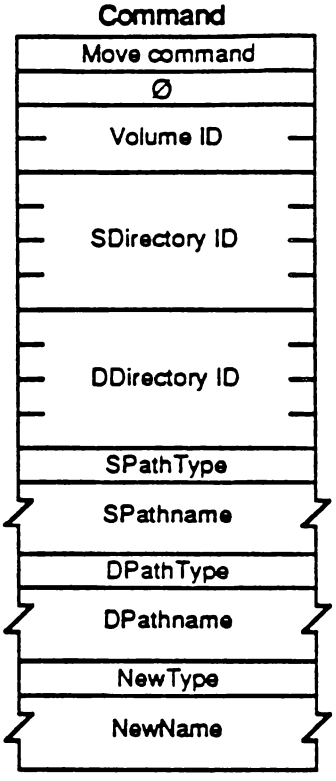
This call is used to move (not just copy) a directory or file to another location on a single volume (source and destination must be on the same volume). An object cannot be moved from one volume to another with this call, even though both volumes may be managed by the server. The destination of the move is specified by providing a DirID and Pathname that indicate the object's new Parent Directory.

INPUTS:	SRefNum (INT)	session refnum
	VolumeID (INT)	volume identifier
	SDirID (LONG)	source ancestor directory identifier
	SPathType (BYTE)	indicates whether SPathname is composed of long names or short names: 1 = all pathname elements are short names 2 = all pathname elements are long names
	SPathname (STR)	pathname of file or directory to be moved (may be null if a directory is to be moved)
	DDirID (LONG)	destination ancestor directory identifier
	DPathType (BYTE)	indicates whether DPathname is composed of long names or short names (same values as SPathType)
	DPathname (STR)	pathname to the destination Parent Directory (may be null)
	NewType (BYTE)	indicates whether NewName is a long name or a short name (same values as SPathType)
	NewName (STR)	new name of file or directory (may be null)
OUTPUTS:	FPError (LONG)	
ERRORS:	ParamErr	unknown session refnum, volume identifier, or pathname type; bad pathname or NewName
	ObjectNotFound	input parameters do not point to an existing file or directory
	ObjectExists	a file or directory with the name NewName already exists
	CantMove	an attempt was made to move a directory into one of its descendent directories
	AccessDenied	user does not have the right to move the file/directory

ALGORITHM: If the object to be moved is a directory, the directory and all its descendents will be moved. The file or directory is moved (deleted from its original Parent Directory) and renamed to its new name. The creation of Long and Short names is performed as described in Appendix B. The object's Mod Date, and the Mod Date of the object's Parent Directory are set to the server's clock. If NewName is null, the object will not be renamed. Its Parent Directory ID will be set to the destination Parent DirID, but all other parameters remain unchanged. The parameters of all descendent directories and files remain unchanged.

RIGHTS: The user must have previously called FPOpenVol for the volume. To move a directory, the user must have Search access rights to all ancestors down to and including the source and destination Parents, as well as Write access right to those directories. To move a file, Search access rights are needed for all ancestors except the source and destination Parents, as well as Read and Write access rights to the source Parent and Write access right to the destination Parent.

PACKET FORMAT:



FPOpenDir

This call is used to open a directory on a Variable-DirID volume and obtain its directory identifier.

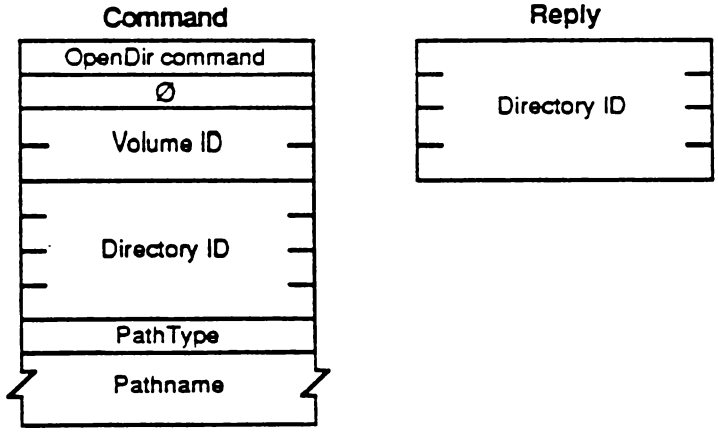
INPUTS:	SRefNum (INT)	session refnum
	VolumeID (INT)	volume identifier
	DirID (LONG)	ancestor directory identifier
	PathType (BYTE)	indicates whether Pathname is composed of long names or short names: 1 = all pathname elements are short names 2 = all pathname elements are long names
	Pathname (STR)	pathname to desired directory (cannot be null)
OUTPUTS:	FPError (LONG)	
	DirID (LONG)	identifier of specified directory
ERRORS:	ParamErr	unknown session refnum, volume identifier, or pathname type; bad pathname
	ObjectNotFound	input parameters do not point to an existing directory
	AccessDenied	user does not have the rights listed below
	ObjectTypeErr	input parameters point to a file

ALGORITHM: If Volume ID specifies a Variable-DirID volume, the server will generate a variable Directory ID for the directory specified by the other input parameters. If the volume is of Fixed-DirID type, the server will return the fixed DirectoryID belonging to this directory.

RIGHTS: The user must have previously called FPOpenVol for this volume. In addition, the user must have Search access rights to all ancestors down to and including this directory's Parent Directory.

NOTES: This call must be issued to obtain a DirID for a directory and to subsequently access the directory on a Variable-DirID volume. It is not considered an error to invoke this call for directories on other types of volumes, although this is not the recommended way to obtain the parameter. Use the FPGetFileDirParms or FPEnumerate calls instead.

PACKET FORMAT:



FPOpenDT

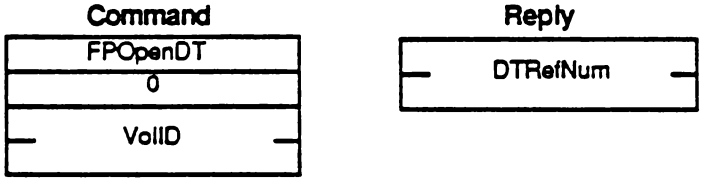
This call is used to retrieve an icon from the Desktop database from a FileCreator/FileType specification.

INPUTS:	DTRefNum (INT)	Desktop Database RefNum
OUTPUTS:	FPError (LONG)	
	DTRefNum (INT)	Refnum for use on future Desktop Manager calls
ERRORS:	ParamErr	unknown session refnum or volume identifier

ALGORITHM: The desktop Database on the selected volume is opened, and a refnum (unique among all volumes on the server) is returned for use in subsequent calls.

RIGHTS: No special rights are needed to make this call.

PACKET FORMAT:



FPOpenFork

This call is used to open the data or resource fork of an existing file for the purpose of reading from it or writing to it. Each fork must be opened separately; a unique OpenForkRefnum will be returned for each.

INPUTS:	SRefNum (INT)	session refnum
	VolumeID (INT)	volume identifier
	DirID (LONG)	ancestor directory identifier
	Bitmap (INT)	bitmap describing which parameters are to be returned (the corresponding bit should be set). This field is the same as that in the FPGetFileDirParms call. (can be null)
	AccessMode (INT)	desired access and sharing modes, as specified by any combination of the following bits: 0 Read - allows the file to be read 1 Write - allows the file to be written to 4 DenyRead - denies others the right to read the fork while this user has it open 5 DenyWrite - denies others the right to write to the fork while this user has it open See Appendix C for a detailed explanation of the use of the Deny bits.
	PathType (BYTE)	indicates whether Pathname is composed of long names or short names 1 = all pathname elements are short names 2 = all pathname elements are long names
	Pathname (STR)	pathname to desired file; cannot be null
	Rsrc/DataFlag (BIT)	flag to indicate which fork is to be opened (all other bits must be zero): 0 = data fork 1 = resource fork
OUTPUTS:	FPError (LONG)	
	Bitmap (INT)	copy of input parameter
	OForkRefnum	refnum used to refer to this fork in subsequent calls
	File Parameters	
ERRORS:	ParamErr	unknown session refnum, volume identifier, or pathname type; null or bad pathname
	ObjectNotFound	input parameters do not point to an existing file
	BitmapErr	an attempt was made to retrieve a parameter which cannot be obtained with this call (file will not be opened)
	DenyConflict	fork cannot be opened because Deny modes conflict (parameters will be returned)
	AccessDenied	user does not have the rights listed below
	ObjectTypeErr	input parameters point to a directory
	TooManyFilesOpen	the server cannot open another fork

ALGORITHM: The server opens the specified fork if the user has the proper access rights for the requested Access Mode, and if the requested Access Mode does not conflict with already-open access paths to this fork.

If the open is successful, the server retrieves the specified parameters for the file and packs them, in Bitmap order, in the reply packet along with a copy of the Bitmap and an OForkRefnum inserted before the parameters. This OForkRefnum is to be used in all subsequent calls involving the open fork.

File parameters will be returned only if the call completes with no error or with a Deny Conflict error. In the latter case, the server will return zero for the OForkRefnum but valid parameters as requested so that the user can determine if he is the one who already has the fork open.

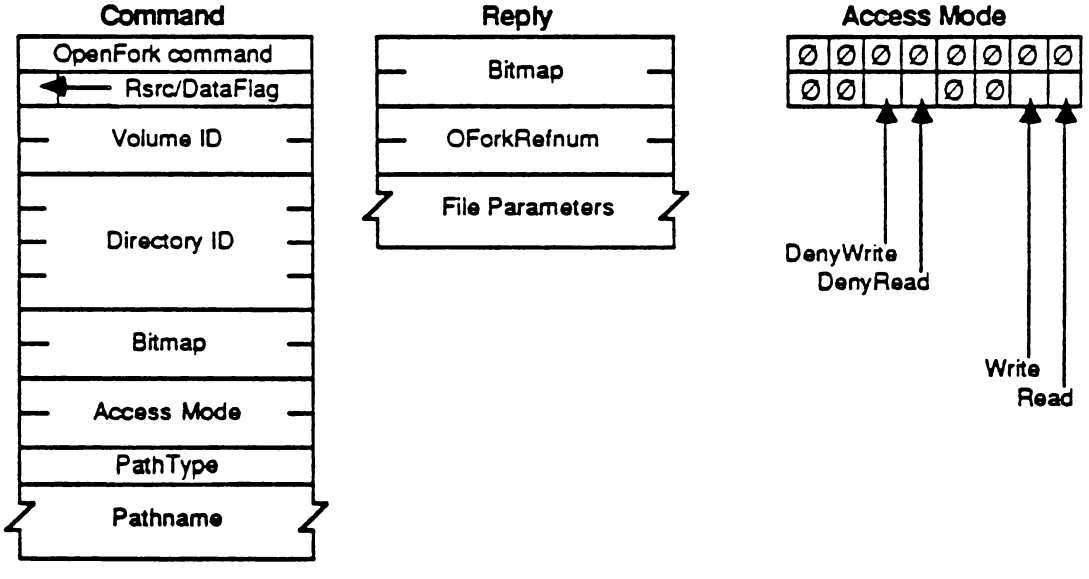
In order to keep all variable-length parameters at the end of the packet (even if more parameters are later added), all such parameters like the Long Name and Short Name fields must be represented in the Bitmap order as fixed-length offsets (INTs) from the start of the parameters (not the start of the bitmap) to the variable-length fields. The actual variable-length fields are then packed after all fixed-length fields.

RIGHTS: The user must have previously called FPOpenVol for this volume. To open a fork for Read, or in the special case in which neither Read nor Write access is requested, the user must have Search access to all ancestors except the Parent, as well as Read access to the Parent.

To open the fork for Write, the volume must not be marked Read Only. If both forks are currently empty, the user must have Search or Write access to all ancestors except the Parent, as well as Write access to the Parent. If either fork is non-empty and one of them is being opened for Write, the user must have Search access to all ancestors except the Parent, as well as Read and Write access to the Parent.

NOTES: If the fork was successfully opened and the user requested the file's Attributes in the Bitmap, the appropriate DAlreadyOpen or RAlreadyOpen bits will be set.

PACKET FORMAT:



FPOpenVol

This call is used to "mount" a volume. It must be called once before any other call can be made to access objects on the volume.

INPUTS:	SRefNum (INT)	session refnum
	Bitmap (INT)	bitmap describing which parameters are to be returned (the corresponding bit should be set). This field is the same as that in the FPGetVolParms call (cannot be null)
	VolumeName (STR)	name of the volume as returned by the FPGetSrvrParms call (maximum 27 bytes)
	Password (8 BYTES)	optional password
OUTPUTS:	FPError (LONG)	
	Bitmap (INT)	copy of input parameter
	Volume Parameters	
ERRORS:	ParamErr	unknown session refnum or volume name
	BitmapErr	an attempt was made to retrieve a parameter which cannot be obtained with this call; null bitmap
	AccessDenied	Password not supplied or does not match

ALGORITHM: The password is sent in the command packet in clear text, padded (suffixed) with null bytes to its full 8-byte length. Password comparison is case sensitive.

If the volume is password-protected, the server will check that the Password supplied by the user matches the one kept with the volume. If they do not match, or if no Password was supplied, an AccessDenied error will be returned.

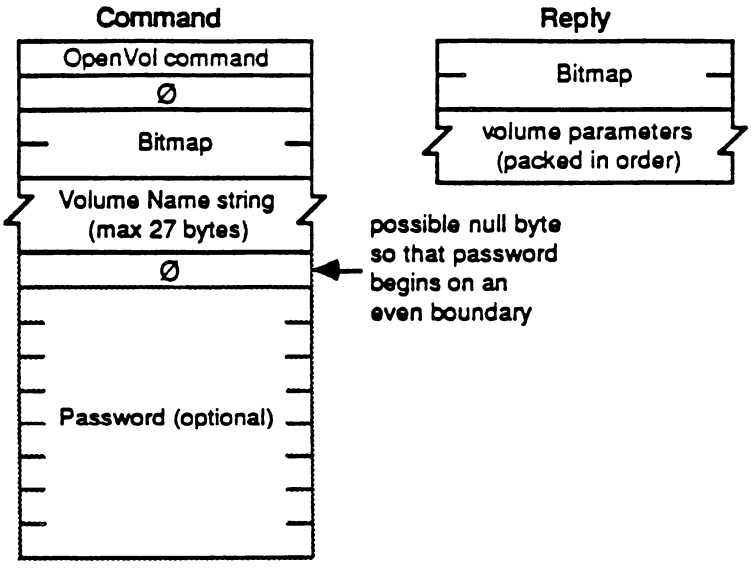
If the Passwords match, or if the volume is not password-protected, the server will retrieve the requested parameters and pack them into the reply packet. The server will mark the volume as "mounted", meaning that this user has permission to make calls relating to objects on this volume.

RIGHTS: No special access rights are needed to make this call.

NOTES: This call cannot be made with a null Bitmap; the Bitmap should at least request the Volume ID to be returned as there is no other way to retrieve this parameter and it is needed for most subsequent calls.

FPOpenVol can be called multiple times without an intervening FPCloseVol call; however, a single FPCloseVol call will invalidate the VolID.

PACKET FORMAT:



FPRead

This call is used to read a block of data from an open fork.

INPUTS:	SRefNum (INT)	session refnum
	OForkRefnum (INT)	open fork refnum
	Offset (LONG)	byte offset from beginning of fork at which the read is to begin
	ReqCount (LONG)	number of bytes to read
	NewlineMask (BYTE)	mask which is ANDed with each character and compared with the NewlineChar to test for the end of the block. In AFP Version 1.1, only two masks are valid. $0= ignore NewlineChar $FF= a NewlineChar is specified
	NewlineChar (BYTE)	any ASCII character from $00 to $FF inclusive which, when encountered in reading the fork (after masking), causes the read operation to terminate
OUTPUTS:	FPError (LONG)	
	ActCount (LONG)	number of bytes actually read from the fork
	Fork data	
ERRORS:	ParamErr	unknown session refnum or open fork refnum; negative ReqCount or Offset; invalid Newline mask
	AccessDenied	fork was not opened for Read
	EOFErr	end-of-fork was reached
	LockErr	some or all of requested range is locked by another user

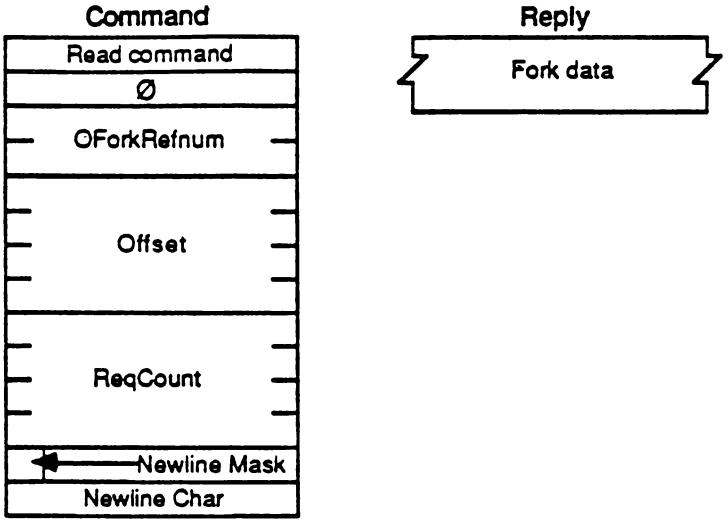
ALGORITHM: The fork is read starting Offset bytes from the beginning of the fork and terminating at the first NewlineChar encountered (if NewlineMask is set to $FF), the end-of-fork, the start of a locked range, or when ReqCount bytes have been read. If the end-of-fork (or the start of a locked range) was reached, all data read up to the fork end (the start of the locked range) will be returned, and an EOFErr (LockErr) will be returned.

Reading a byte that was never written to the fork will return an undefined value.

RIGHTS: The fork must have been opened for Read.

NOTES: Locking of the range should be done prior to this call since the underlying transport mechansim may force the request to be broken up into multiple smaller requests. Although the range may not be locked when the call begins execution, it is possible for another user to lock some or all of the range before this call completes, causing the Read to succeed partially.

PACKET FORMAT:



FPRemoveAPPL

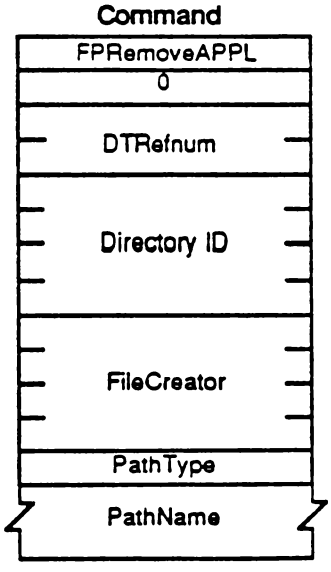
This call is used to remove an APPL mapping from the Desktop Database.

INPUTS:	SRefNum (INT)	session refnum
	DTRefnum (INT)	Desktop Database refnum
	DirID (LONG)	Directory identifier
	FileCreator (RESTYPE)	Creator type of application to be removed
	PathType (BYTE)	indicates whether Pathname is composed of long names or short names 1 = all pathname elements are short names 2 = all pathname elements are long names
	PathName (STRING)	pathname to the application being removed
OUTPUTS:	FPError (LONG)	
ERRORS:	ParamErr	unknown session refnum or Desktop Database refnum
	ObjectNotFound	input parameters do not point to an existing file
	AccessDenied	user does not have the rights listed below
	ItemNoFound	no APPL entry corresponding to the input parameters was found in the Desktop Database

ALGORITHM: The entry for the application specified is located in the Desktop /Database using the FileCreator information. If an entry is found for the specified Directory ID/File name, the entry is removed.

RIGHTS: The user must have previously called FPOpenDT for the corresponding volume. In addition, the file must be present in the specified directory before this call is issued. The user must have Search access to all ancestors except the Parent, as well as Read and Write access to the Parent.

PACKET FORMAT:



FPRemoveComment

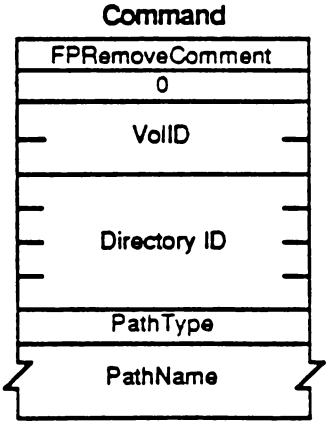
This call is used to remove a comment from the Desktop Database

INPUTS:	SRefNum (INT)	session refnum
	DTRefnum (INT)	Desktop Database refnum
	DirID (LONG)	directory identifier
	PathType (BYTE)	indicates whether Pathname is composed of long names or short names 1 = all pathname elements are short names 2 = all pathname elements are long names
	PathName (STRING)	the pathname for the file or folder associated with the comment
OUTPUTS:	FPError (LONG)	
ERRORS:	ParamErr	unknown session refnum, volume identifier, or pathname type; bad pathname
	ItemNotFound	no comment found in Desktop Database
	AccessDenied	user does not have the rights listed below
	ObjectNotFound	input parameters do not point to an existing object

ALGORITHM: The comment associated with the file or folder specified is removed from the Desktop Database.

RIGHTS: The user must have previously called FPOpen DT for the corresponding volume. If the comment is associated with a non-empty directory, the user must have Search access to all ancestors including the Parent Directory, plus Write access to the Parent. If the comment is associated with an empty directory, the user must have Search or Write access to all ancestors including the Parent Directory, plus Write access to the Parent. If the comment is associated with a non-empty file, the user must have Search access to all ancestors except the Parent Directory, plus Read and Write access to the Parent. If the comment is associated with an empty file, the user must have Search or Write access to all ancestors except the Parent Directory, plus Write access to the Parent.

PACKET FORMAT:



FPRename

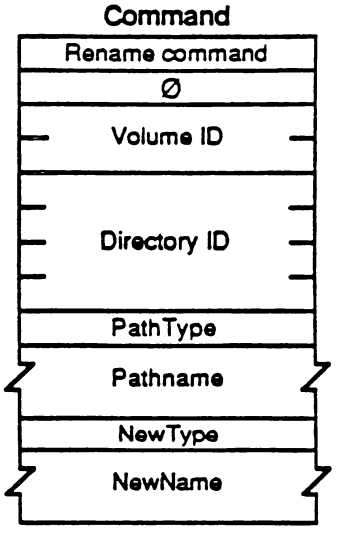
This call is used to rename either a directory or a file.

INPUTS:	SRefNum (INT)	session refnum
	VolumeID (INT)	volume identifier
	DirID (LONG)	ancestor directory identifier
	PathType (BYTE)	indicates whether Pathname is composed of long names or short names. 1 = all pathname elements are short names 2 = all pathname elements are long names
	Pathname (STR)	pathname of file or directory to be renamed (may be null if a directory is to be renamed)
	NewType (BYTE)	indicates whether NewName is a long name or short name, (same values as PathType)
	NewName (STR)	new name of file or directory (cannot be null)
OUTPUTS:	FPError (LONG)	
ERRORS:	ParamErr	unknown session refnum, volume identifier, or pathname type; bad pathname or NewName
	ObjectNotFound	input parameters do not point to an existing file or directory
	ObjectExists	a file or directory with the name NewName already exists
	AccessDenied	user does not have the right to rename the file/directory
	CantRename	an attempt was made to rename a volume

ALGORITHM: The object is renamed to its new name. The creation of Long and Short names is performed as described in Appendix B. The Mod Date of the Parent Directory is set to the server's clock.

RIGHTS: The user must have previously called FPOpenVol for the volume. In addition, the user must have Search access rights to all ancestors except the object's Parent Directory, as well as Write access right to the Parent directory. To rename a directory, the user must also have Search access to the Parent; else to rename a file, the user must also have Read access to the Parent.

PACKET FORMAT:



FPSetDirParms

This call is used to set parameters for a particular directory.

INPUTS:	SRefNum (INT)	session refnum
	VolumeID (INT)	volume identifier
	DirID (LONG)	ancestor directory identifier
	Bitmap (INT)	bitmap describing which parameters are to be set (the corresponding bit should be set). This field is the same as that in the FPGetFileDirParms call.
	PathType (BYTE)	indicates whether Pathname is composed of long names or short names: 1 = all pathname elements are short names 2 = all pathname elements are long names
	Pathname (STR)	pathname to desired directory
	Directory Parameters	
OUTPUTS:	FPError (LONG)	
ERRORS:	ParamErr	unknown session refnum, volume identifier, or pathname type; bad pathname; owner or group ID not valid
	ObjectNotFound	input parameters do not point to an existing directory
	BitmapErr	an attempt was made to set a parameter which cannot be set with this call; null bitmap
	AccessDenied	user does not have the rights listed below
	ObjectTypeErr	input parameters point to a file

ALGORITHM: The server sets the specified parameters for the directory which must be packed, in Bitmap order, in the command packet. In order to keep all variable-length parameters at the end of the packet (even if more parameters are later added), all such parameters like the Long Name and Short Name fields must be represented in the Bitmap order as fixed-length offsets (INTs) from the start of the parameters to the start of the variable-length fields. The actual variable-length fields are then packed after all fixed-length fields.

Changing a directory's AccessRights will immediately affect other currently open sessions.

If the Access Controls or Owner ID or Group ID are set in this call, and the Mod Date is not set, the Mod Date will be set to the the server's clock.

RIGHTS: The user must have previously called FPOpenVol for this volume. To set a directory's Access Controls, Owner ID, or Group ID, the user must have Search or Write access rights to all ancestors including this directory's Parent Directory, and must be the owner of the directory. To set any other parameter for an empty directory, the user must have Search or Write access to all ancestors except the Parent, as well as Write access to the Parent. To set any other parameter for a non-empty directory, the user must have Search access to all ancestors including the Parent, as well as Write access to the Parent.

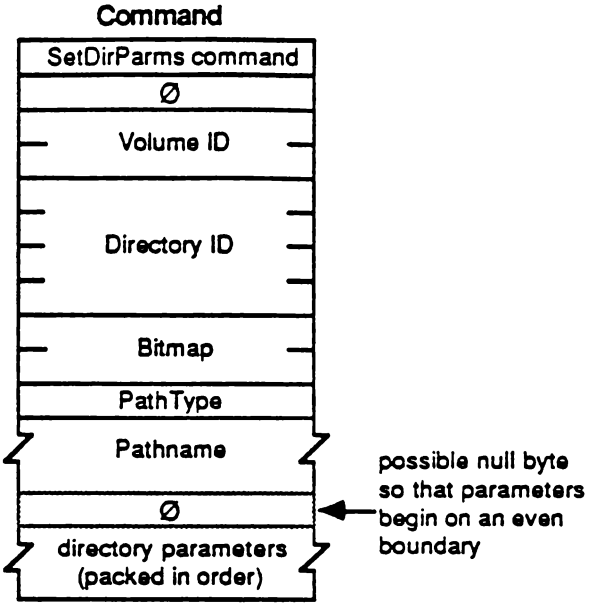
If the user lacks the access rights to set any one of a number of parameters, an Access Denied error will be returned and no parameters will be set.

NOTES: This call cannot be used to set a directory 's name (use FPRename), Parent Directory (use FPMove), Directory ID, or number of offspring.

The user needs to be the owner of the directory to set the directory's Access Controls, Owner ID, or Group ID only.

A null byte may be added between the Pathname and the Directory Parameters so that the Parameters begin on an even boundary in the Command Buffer.

PACKET FORMAT:



FPSetFileParms

This call is used to set parameters for a particular file.

INPUTS:	SRefNum (INT)	session refnum
	VolumeID (INT)	volume identifier
	DirID (LONG)	ancestor directory identifier
	Bitmap (INT)	bitmap describing which parameters are to be set (the corresponding bit should be set). This field is the same as that in the FPGetFileDirParms call.
	PathType (BYTE)	indicates whether Pathname is composed of long names or short names: 1 = all pathname elements are short names 2 = all pathname elements are long names
	Pathname (STR)	pathname to desired file
	File Parameters	
OUTPUTS:	FPError (LONG)	
ERRORS:	ParamErr	unknown session refnum, volume identifier, or pathname typ null or bad pathname
	ObjectNotFound	input parameters do not point to an existing file
	AccessDenied	user does not have the right to access the file
	BitmapErr	an attempt was made to set a parameter which cannot be set with this call; null bitmap
	ObjectTypeErr	input parameters point to a directory

ALGORITHM: The server sets the specified parameters for the file which must be packed, in Bitmap order, in the command packet. In order to keep all variable-length parameters at the end of the packet (even if more parameters are later added), all such parameters must be represented in the Bitmap order as fixed-length offsets (INTs) from the start of the parameters to the start of the variable-length fields. The actual variable-length fields are then packed after all fixed-length fields.

In AFP Version 1.1, only the following parameters may be set or cleared: Attributes (all except DAlreadyOpen and RAlready Open), Create Date, Last Mod Date, Backup Date, and FinderInfo.

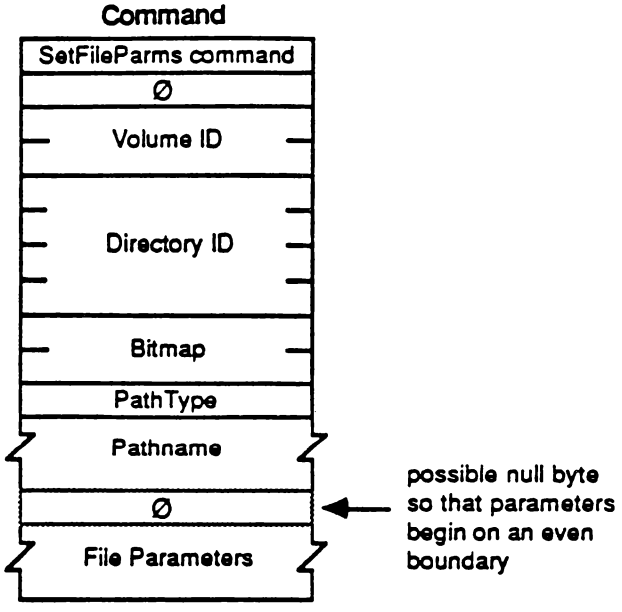
If the Attributes field is included, the Set/Clear bit is used to indicate that the specified Attributes (whose corresponding bits are set) are to be either set or cleared (0 = clear specified Attributes, 1 = set specified Attributes). Hence it is not possible to set some Attributes and clear others in the same call.

RIGHTS: The user must have previously called FPOpenVol for this volume. If the file is empty (both forks are of zero length), the user must have Search or Write access rights to all ancestors except this file's Parent Directory, as well as Write access right to the Parent Directory. If either fork is non-empty, the user must have Search access to all ancestors except the Parent, as well as Read and Write access to the Parent.

NOTES: This call cannot be used to set a file's name (use FPRename), Parent Directory (use FPMove), File Number, or fork lengths.

A null byte may be added between the Pathname and the File Parameters so that the Parameters begin on an even boundary in the Command Buffer.

PACKET FORMAT:



FPSetFileDirParms

This call is used to set parameters for a particular object, either a file or a directory.

INPUTS:	SRefNum (INT)	session refnum
	VolumeID (INT)	volume identifier
	DirID (LONG)	ancestor directory identifier
	Bitmap (INT)	bitmap describing which parameters are to be set (the corresponding bit should be set). This field is the same as the FileBitmap or DirBitmap in the FPGetFileDirParms call (only the parameters which are common to both bitmaps may be set by this call).
	PathType (BYTE)	indicates whether Pathname is composed of long names or short names, or unknown: 1 = all pathname elements are short names 2 = all pathname elements are long names
	Pathname (STR)	pathname to desired object
	Parameters	
OUTPUTS:	FPError (LONG)	
ERRORS:	ParamErr	unknown session refnum, volume identifier, or pathname type; bad pathname
	ObjectNotFound	input parameters do not point to an existing object
	AccessDenied	user does not have the rights listed below
	BitmapErr	an attempt was made to set a parameter which cannot be set with this call; null bitmap

ALGORITHM: In AFP Version 1.1, the only parameters that may be set or cleared by this call are: Attributes (only the Invisible attribute), Create Date, Last Mod Date, Backup Date, and FinderInfo. Note that these parameters are common to both files and directories. The server sets the specified parameters for the object which must be packed, in Bitmap order, in the command packet. In order to keep all variable-length parameters at the end of the packet (even if more parameters are later added), all such parameters must be represented in the Bitmap order as fixed-length offsets (INTs) from the start of the parameters to the start of the variable-length fields. The actual variable-length fields are then packed after all fixed-length fields.

If the Attributes field is included, the Set/Clear bit is used to indicate that the specified Attributes (whose corresponding bits are set) are to be either set or cleared (0 = clear specified Attributes, 1 = set specified Attributes). Hence it is not possible to set some Attributes and clear others in the same call.

RIGHTS: The user must have previously called FPOpenVol for this volume. To set the parameters for a non-empty directory, the user needs Search access to all ancestors including the Parent Directory, as well as Write access to the Parent. To set the parameters for an empty directory, the user needs Search or Write access to all ancestors except the Parent Directory, as well as Write access to the Parent.

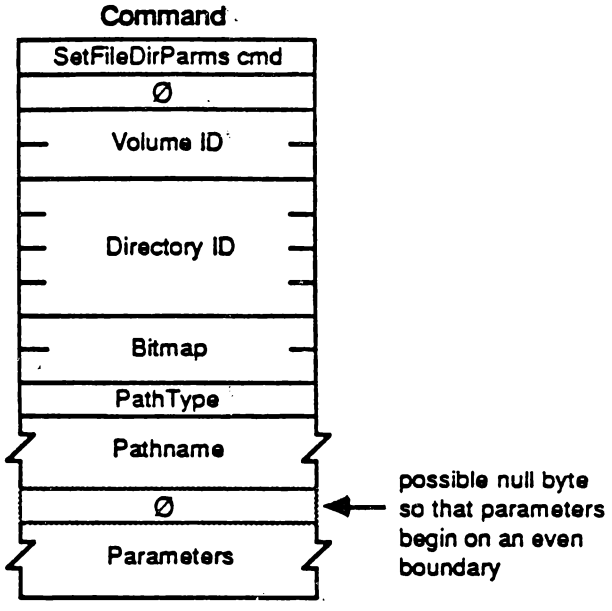
To set the parameters for a non-empty file, the user needs Search access to all ancestors except the Parent Directory, as well as Read and Write access to the Parent. To set the parameters for an empty file, the user needs Search or Write access to all ancestors except the Parent Directory, as well as Write access to the Parent.

NOTES: The Create Date, Mod Date, Backup Date, and FinderInfo may be set with the FPSetFileParms or FPSetDirParms calls if the user knows that the object is a file or a directory. To set a directory's Access Rights, Creator ID, or Group ID, use the

FPSetDirParms call. To set a file's Attributes other than Invisible, use the FPSetFileParms call.

A null byte may be added between the Pathname and the Parameters so that the Parameters begin on an even boundary in the Command Buffer.

PACKET FORMAT:



FPSetForkParms

This call is used to set parameters for a file associated with a particular open fork.

INPUTS:	SRefNum (INT)	session refnum
	OForkRefnum (INT)	open fork refnum
	Bitmap (INT)	bitmap describing which parameters are to be set (the corresponding bit should be set). This field is the same as that in the FPGetFileDirParms call; however, in AFP Version 1.1, only the fork length may be set.
	Fork Length	
OUTPUTS:	FPError (LONG)	
ERRORS:	ParamErr	unknown session refnum or open fork refnum
	BitmapErr	an attempt was made to set a parameter which cannot be set with this call; null bitmap
	DiskFull	no more space on the volume
	LockErr	locked range conflict
	AccessDenied	fork was not opened for Write

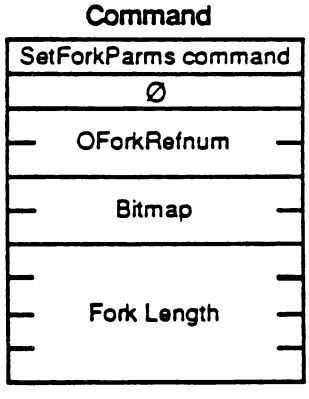
ALGORITHM: The Bitmap and fork length are passed to the server, which changes the length of the fork specified by OForkRefnum. A BitmapErr will be returned if an attempt is made to set the length of the other fork comprising the file, or any other file parameter.

A LockErr will be returned if one of the following conditions occurs: a) the user attempts to set the end-of-fork into a range locked by another user, B) the file is truncated so as to eliminate a range locked by another user, c) the range between the old end-of-fork and the new end-of-fork intersects any lock owned by another user.

RIGHTS: The fork must have been opened for Write.

NOTES: This call cannot be used to set a file's name (use FPRename), Parent Directory (use FPMove), or File Number.

PACKET FORMAT:



FPSetVolParms

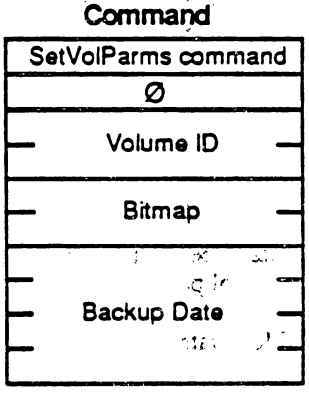
This call is used to set the parameters for a particular volume. The volume is specified by its VolumeID as returned from the FPOpenVol call.

INPUTS:	SRefNum (INT)	session refnum
	VolumeID (INT)	volume identifier
	Bitmap (INT)	bitmap describing which parameters are to be set (the corresponding bit should be set). This field is the same as that in the FPGetVolParms call; however, in AFP Version 1.1, only the Backup Date field may be set.
	Backup Date (LONG)	new Backup Date
OUTPUTS:	FPError (LONG)	
ERRORS:	ParamErr	unknown session refnum or volume identifier
	BitmapErr	an attempt was made to set a parameter which cannot be set with this call; null bitmap

ALGORITHM: The Bitmap and Backup Date are passed to the server, which changes the date of the specified volume.

RIGHTS: The user must have previously called FPOpenVol for this volume.

PACKET FORMAT:



FPWrite

This call is used to write a block of data to an open fork.

INPUTS:	SRefNum (INT)	session refnum
	OForkRefnum (INT)	open fork refnum
	Offset (LONG)	byte offset from the beginning or end of the fork at which the write is to begin (should be negative to indicate a point within the fork relative to the end)
	ReqCount (LONG)	number of bytes to write
	Start/EndFlag (BIT)	one-bit flag indicating whether the Offset field is relative to the beginning or end of the fork (all other bits must be zero): 0 = relative to the beginning of the fork 1 = relative to the end of the fork
	Fork data	
OUTPUTS:	FPError (LONG)	
	ActCount (LONG)	number of bytes actually written to the fork
	LastWritten (LONG)	the number of the byte just past the last byte written
ERRORS:	ParamErr	unknown session refnum or open fork refnum
	AccessDenied	fork was not opened for Write
	LockErr	some or all of requested range is locked by another user
	DiskFull	no more space on the volume

ALGORITHM: The fork is written starting Offset bytes from the beginning or end of the fork. If the write extends beyond the end-of-fork, the fork is extended. If part of the range is locked by another user, a LockErr will be returned and no data will be written.

If multiple writers are concurrently modifying the fork, they may each have a different notion of the end-of-fork, although the server always knows the correct end-of-fork. It is for this reason that the "write relative to end-of-fork" feature is provided. The number of the byte just past the last byte written is returned, since the end-of-fork may have been different from the user's notion at the time at which the write was made.

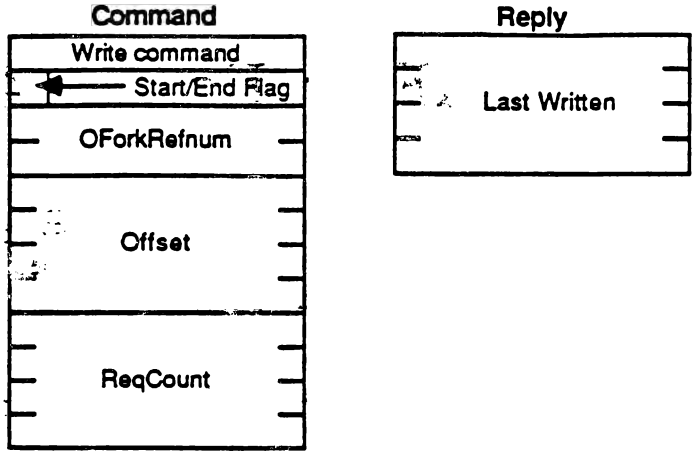
The fork data to be written is transmitted to the server in an intermediate exchange of Session Protocol packets.

The file's Mod Date is not changed until the fork is closed.

RIGHTS: The fork must have been opened for Write.

NOTES: Locking the range before writing to it is highly recommended, since the underlying transport mechanism may force the request to be broken up into multiple small requests. Although the range may not be locked when the call begins execution, it is possible for another session to lock some or all of the range before this call completes, causing the write to succeed partially.

PACKET FORMAT:



Chapter 8

AFP's Use of ASP

The AppleTalk Filing Protocol uses the transport services of the AppleTalk Session Protocol (ASP). This chapter discusses the manner in which the AFP calls are conveyed via ASP. For this purpose we refer to various terms defined in the ASP specification document. For these definitions, the reader must turn to that document.

In the case of all these calls, the AFP level variable *FPError* is returned as the ASP-level *CmdResult*. Also, the AFP-level Command packet is conveyed as an ASP-level *Command block*, while the AFP-level Reply packet is returned as the ASP-level *Command Reply block*.

Finding a Server

To find an AFP server, an NBP request must be issued for objects of type "AFPServer".

Getting Server Information

A workstation AFP client makes an *FPGetSrvrInfo* call to obtain server information needed before an AFP level login can be attempted. The *FPGetSrvrInfo* call is converted by the workstation's AFP into an ASP level *SPGetStatus* call. The server information is returned by ASP as the *SPGetStatus* call's *Command reply block*. Note that the *SAddr* (internet address of the file server's SLS) supplied with the *FPGetSrvrInfo* call is used as the *SLSEntityIdentifier* required by the *SPGetStatus* call.

Login On the File Server

Having obtained the server information, the AFP client initiates the process for logging in on the file server. This is achieved by AFP in a two step fashion.

First, AFP issues an *SPOpenSession* call to establish an ASP-level session. If for some reason this ASP session can not be established, then an error is returned to the workstation's AFP client and login is abandoned.

Once the ASP session has been opened, then the workstation's AFP issues an *SPCommand* call to send the FPLogin's *Command* to the file server's AFP. This FPLogin Command is sent as the *Command block* of the SPCommand. Beyond this point a series of SPCommand's may be necessary to complete the login process (this depends on the User Authentication Method being employed in the login).

If at any point, in this process the server returns an error indicating that the login can not be completed successfully, then the workstation's AFP must make an *SPCloseSession* call to close the ASP session. This is vital, otherwise the session will remain open for no purpose. After closing the session, the workstation's AFP returns an appropriate error message to its client.

Logout of the File Server

When the workstation client wishes to terminate its conversation with the file server, it issues an *FPLogout* call to its AFP. The workstation's AFP again uses a two step process to carry out this logout.

First, it issues an *SPCommand* to convey the *FPLogout* command to the server. Having done so, it then terminates the corresponding ASP session by making an *SPCloseSession* call.

Other AFP Calls

Every other AFP calls (with the exception of *FPWrite*) is conveyed by AFP via an *SPCommand* call.

An *FPWrite* call is sent by the workstation's AFP by making an *SPWrite* call.

It should be noted that in the case of the *FPRead*, *FPWrite*, and *FPEnumerate* calls, a partial completion of the call is possible. By this we mean that less than the desired length is read (or written). There are two reasons for this.

First, in the case of an *FPRead* call, the end-of-file may be reached before the requested number of bytes have been read. In the case of an *FPRead* or *FPWrite* call, a locked byte range may be reached before all the requested bytes have been read or written.

A second reason is related to the *QuantumSize* related to the session protocol. In the case of AppleTalk, the largest data block that may be written or read through ASP is equal to 4624 bytes, the *QuantumSize* size. Thus the underlying *SPCommand* (in the case of an *FPRead* or *FPEnumerate*) or *SPWrite* (in the case of an *FPWrite*) will complete with an actual received or written reply size smaller than the requested value. Therefore, if no error was returned, the AFP must issue an additional *SPCommand* or *SPWrite* call to complete the AFP request.

Although AFP may have to issue several ASP calls to complete a single AFP command, the first ASP command should convey the actual sizes requested by the user to allow the server a chance to optimize. Subsequent ASP calls should convey sizes that have been adjusted to reflect how much of the original command has already been completed.

Appendix A
User Authentication Methods

As noted in the main body of this document, AFP version 1.1 provides three standard user authentication methods. These correspond to UAM strings "NoUserAuthent", "Cleartxt passwrd" and "Randnum exchange". [Note: these strings should be used in a case-insensitive fashion].

No User Authentication

The first of these in fact corresponds to no user authentication (UAM = "No User Authent") and as such needs no specification. Thus, no user name or password information is required in the *FPLogin* command which therefore has no *UserAuthInfo* field.

Any server can accept a user login using this method. If the server handles the user-authentication-based directory-level access control mechanism, then it must assign the user a special User ID and Group ID for that session, such that the user only obtains world's access rights for every directory in every server volume.

User Authentication with Clear Text Password Transmission

The second standard method employs the transmission of the user's password in clear text (in addition to the user's name) in the *FPLogin* command packet. The UserAuthInfo part of the *FPLogin* command consists of the user's name (a string of up to 31 characters which follows the UAM field in the packet without padding) followed by a possible null byte and then the user's password. The *user's password* is an 8-byte quantity. If the user provides a shorter password then it must be padded (suffixed) with null ($00) bytes to make its length equal to 8 bytes. The permissible set of characters in passwords consists of all 8-bit characters with the most significant bit equal to 0.

User name comparison in servers must be case insensitive, but password comparison is to be case sensitive in this UserAuthenticationMethod. Of course, one could create a new UserAuthenticationMethod that performed case-insensitive password comparison.

It should be noted that this method should be used by workstations only if it is known that the intervening network is secure against "wire tapping", otherwise the password information can be picked up out of *FPLogin* command packets by anyone tapping the network.

User Authentication Based on Random Number Exchange

GetIconInfo	52
AddAPPL	53
RmvAPPL	54
GetAPPL	55
AddComment	56
RmvComment	57
GetComment	58
AddIcon	192

Appendix A

User Authentication Methods

As noted in the main body of this document, AFP version 1.1 provides three standard user authentication methods. These correspond to UAM strings "NoUserAuthent", "Cleartxt passwrd" and "Randnum exchange". [Note: these strings should be used in a case-insensitive fashion].

No User Authentication

The first of these in fact corresponds to no user authentication (UAM = "No User Authent") and as such needs no specification. Thus, no user name or password information is required in the *FPLogin* command which therefore has no *UserAuthInfo* field.

Any server can accept a user login using this method. If the server handles the user-authentication-based directory-level access control mechanism, then it must assign the user a special User ID and Group ID for that session, such that the user only obtains world's access rights for every directory in every server volume.

User Authentication with Clear Text Password Transmission

The second standard method employs the transmission of the user's password in clear text (in addition to the user's name) in the *FPLogin* command packet. The UserAuthInfo part of the *FPLogin* command consists of the user's name (a string of up to 31 characters which follows the UAM field in the packet without padding) followed by a possible null byte and then the user's password. The *user's password* is an 8-byte quantity. If the user provides a shorter password then it must be padded (suffixed) with null ($00) bytes to make its length equal to 8 bytes. The permissible set of characters in passwords consists of all 8-bit characters with the most significant bit equal to 0.

User name comparison in servers must be case insensitive, but password comparison is to be case sensitive in this UserAuthenticationMethod. Of course, one could create a new UserAuthenticationMethod that performed case-insensitive password comparison.

It should be noted that this method should be used by workstations only if it is known that the intervening network is secure against "wire tapping", otherwise the password information can be picked up out of *FPLogin* command packets by anyone tapping the network.

User Authentication Based on Random Number Exchange

In environments where the network is not secure against tapping, a more secure method based on a random number exchange between the server and the workstation can be used. In this method, the user's password is never sent over the network and hence cannot be picked up by tapping. In fact, it is essentially impossible (as secure as the basic encryption method) to derive the password from the information that is sent over the network.

New Open Attempt Deny Mode and Access Mode

Deny R/W	Deny R/W					Deny Write				Deny Read				Deny None			
		-	R	RW	W	-	R	RW	W	-	R	RW	W	-	R	RW	W
	-	√				√				√				√			
	R					√								√			
	RW													√			

Error Codes

Each call returns an Error code which is a 4-byte integer. The various error values are listed below together with their mnemonic names (these are the names used in Chapter 7). The values given below are in hexadecimal and decimal (base-10) form.

Error Mnemonic	Hex Value	Decimal Value
NoErr	$0	0
AccessDenied	$FFFFEC78	-5000
AuthContinue	$FFFFEC77	-5001
BadUAM	$FFFFEC76	-5002
BadVersNum	$FFFFEC75	-5003
BitmapErr	$FFFFEC74	-5004
CantMove	$FFFFEC73	-5005
DenyConflict	$FFFFEC72	-5006
DirNotEmpty	$FFFFEC71	-5007
DiskFull	$FFFFEC70	-5008
EOFErr	$FFFFEC6F	-5009
FileBusy	$FFFFEC6E	-5010
FlatVol	$FFFFEC6D	-5011
ItemNotFound	$FFFFEC6C	-5012
LockErr	$FFFFEC6B	-5013
MiscErr	$FFFFEC6A	-5014
NoMoreLocks	$FFFFEC69	-5015
NoServer	$FFFFEC68	-5016
ObjectExists	$FFFFEC67	-5017
ObjectNotFound	$FFFFEC66	-5018
ParamErr	$FFFFEC65	-5019
RangeNotLocked	$FFFFEC64	-5020
RangeOverlap	$FFFFEC63	-5021
SessClosed	$FFFFEC62	-5022
UserNotAuth	$FFFFEC61	-5023
CallNotSupported	$FFFFEC60	-5024
ObjectTypeErr	$FFFFEC5F	-5025
TooManyFilesOpen	$FFFFEC5E	-5026
ServerGoingDown	$FFFFEC5D	-5027
CantRename	$FFFFEC5C	-5028
DirNotFound	$FFFFEC5B	-5029
IconTypeError	$FFFFEC5A	-5030

Appendix E
List of Abbreviations

AFI	AppleTalk filing interface
AFP	AppleTalk Filing Protocol
ALAP	AppleTalk Link Access Protocol
ASP	AppleTalk Session Protocol
ATP	AppleTalk Transaction Protocol
CAM	Current Access Mode
CDM	Current Deny Mode
DDP	Datagram Delivery Protocol
GMT	Greenwich Mean Time
HFS	Macintosh's hierarchical file system
MFS	the earlier Macintosh (flat) file system
NBP	Name Binding Protocol
NFI	Native filing interface
SLS	Session listening socket
UAM	User authentication method

Appendix A

User Authentication Methods

As noted in the main body of this document, AFP version 1.1 provides three standard user authentication methods. These correspond to UAM strings "NoUserAuthent", "Cleartxt passwrd" and "Randnum exchange". [Note: these strings should be used in a case insensitive fashion].

No User Authentication

The first of these in fact corresponds to no user authentication (UAM = "No User Authent") and as such needs no specification. Thus, no user name or password information is required in the *FPLogin* command which therefore has no *UserAuthInfo* field.

Any server can accept a user login using this method. If the server handles the user-authentication-based directory-level access control mechanism, then it must assign the user a special User ID and Group ID for that session, such that the user only obtains world's access rights for every directory in every server volume.

User Authentication with Clear Text Password Transmission

The second standard method employs the transmission of the user's password in clear text (in addition to the user's name) in the *FPLogin* command packet. The UserAuthInfo part of the *FPLogin* command consists of the user's name (a string of up to 31 characters which follows the UAM field in the packet without padding) followed by a possible null byte and then the user's password. The *user's password* is an 8-byte quantity. If the user provides a shorter password then it must be padded (suffixed) with null ($00) bytes to make its length equal to 8 bytes. The permissible set of characters in passwords consists of all 8-bit characters with the most significant bit equal to 0.

User name comparison in servers must be case insensitive, but password comparison is to be case sensitive in this UserAuthenticationMethod. Of course, one could create a new UserAuthenticationMethod that performed case-insensitive password comparison.

It should be noted that this method should be used by workstations only if it is known that the intervening network is secure against "wire tapping", otherwise the password information can be picked up out of *FPLogin* command packets by anyone tapping the network.

User Authentication Based on Random Number Exchange

In environments where the network is not secure against tapping, a more secure method based on a random number exchange between the server and the workstation can be used. In this method, the user's password is never sent over the network and hence cannot be picked up by tapping. In fact, it is essentially impossible (as secure as the basic encryption method) to derive the password from the information that is sent over the network.

The underlying idea of the method is that the server knows the user's password and it wants to find out if the user trying to log in knows this password as well. In a sense, the method tries to determine if the user and the server "share the same secret", the password, without sending the secret information over the network.

First, the user sends the FPLogin command packet with UAM corresponding to "Randnum exchange" and the UserAuthInfo containing the user's name string, which follows the UAM in the packet without padding.

Upon receiving this packet, the server examines its user data base to determine if this is a valid user name. If the user name is not found in the user data base, then an error is sent to the workstation indicating this result and login is denied. If the name is found in the user data base, then the server generates a random number (64 bits in length) and sends it back to the workstation. This is returned as the reply to the *FPLogin* call, along with an ID number and an *AuthContinue* error. Although not really an "error", this value is returned to indicate that all is well so far but the user is not yet authenticated.

The workstation uses the NBS data encryption standard (DES) algorithm to encrypt the random number, using the user's password (case-sensitive, same format as in the clear text UAM) as the encryption key. It sends the encrypted value (64 bits) back to the server in the UserAuthInfo parameter of the *FPLoginCont* call along with the ID number returned from the FPLogin call. This ID number merely helps the server associate the two calls and is not used again. The server compares the workstation's encrypted value with the encrypted value obtained using the password from its user data base. If the two encrypted values match, then the user is considered to have been authenticated and the login is successful. The FPLoginCont call will return *NoErr* if so, *UserNotAuth* if not. In either case, no reply data is returned.

Appendix B

Long/Short Name Management Algorithms

Files and directories in AFP possess two names, a Long Name and a Short Name, because of differences in Macintosh and MS-DOS naming rules: Macintosh permits file and directory names to be made up of at most 32 characters, where valid characters may be any printable ASCII code except colon ($3A). MS-DOS is more restrictive; names may be up to eight alphanumeric (plus other) characters, optionally followed by period ($2E) and a one-to-three alphanumeric (plus other) character extension. The dual naming convention was devised to allow file servers to present to Macintosh and MS-DOS users a name space that is compatible with each of their respective file systems. Yet in order to ensure that an object can be uniquely specified by either name, some rules must be adhered to.

AFP naming rules are such that any MS-DOS name can be used directly as an object's Short Name, and any Macintosh name can be used directly as a Long Name. It is up to the file server to generate the "other" name for each object, deriving it from the first and making the name as "close" as possible to minimize semantic loss. Since Long Name format is a superset of Short Name format, some cases are easily handled. For example, if an MS-DOS machine generates a Short Name for an object, the object's Long Name can be made the same as the Short Name. Deriving a Short Name from a Long Name is not as simple, and AFP does not stipulate an exact algorithm for doing this, so that different servers may perform it differently. However, it is crucial that no two objects in a given directory have the same Short Name or the same Long Name. This ensures that any object can be uniquely specified by either name.

Below are the algorithms intended for use by file servers for creating and maintaining a consistent dual name space.

When an object is created (either by *FPCreateFile* or *FPCreateDir*), the caller supplies the object's name and a name type that indicates whether the name is of Short format or Long format. The server must of course first check the name itself to verify that it conforms to the specified format. The algorithm below describes how servers are to assign Short and Long names to objects:

```
IF name type is Short OR name is in Short format
THEN        check for new name in list of short names
        IF name already exists
        THEN        return ObjectExists error
        ELSE set object's Short and Long names to new name

ELSE { name type is Long OR name is in Long format }
        check for new name in list of long names
        IF name already exists
        THEN        return ObjectExists error
        ELSE set object's Long Name to new name
                derive Short Name from Long Name
```

This algorithm is to be used for renaming as well (*FPRename*). When an object is renamed, both names will be changed using the above rules.

Note that this algorithm mandates that whenever an object is named with a Short Name format name, the Long Name will always match. This is done so that servers need only check one list, Long Names or Short Names, for each creation or renaming operation.

An interesting problem can arise. Consider the case in which a server contains a Macintosh file with a Long Name *MacFileWithLongName* and a Short Name *MacFile* which the server derived from the Long-Name. This Short Name is not normally visible to Macintosh workstations. Now suppose that a user at a Macintosh decides to create a new file in the same directory with the Long Name *MacFile*. The call must fail, because by the above algorithm the new file's Short Name and Long Name must both be set to *MacFile* since it is in Short format. Hence the user is returned an ObjectExists error even though the directory does not appear to contain a file by that Long Name.

Appendix C

File Sharing Modes

AFP provides much functionality to control the sharing of files, primarily at the Creator/Group/World Access Rights level but also at the time of file fork open. To perform the latter function, the server must enforce the synchronization rules presented below.

When a fork of a file is opened, the user indicates what kind of Access Mode is needed: *Read*, *Write*, *Read/Write*, or *None* (the latter allows no further access to the fork except to close it, yet it is included since it may be useful as a synchronization primitive). In addition, the user indicates to the server a Deny Mode: just what rights should be denied to others trying to open the fork while the current users have it open. Users that subsequently try to open the fork could be denied *None*, *Read*, *Write*, or *Read/Write* access.

An *FPOpenFork* call could fail for several reasons:
1) The user may not possess the correct rights as Owner/Group/World to allow the desired access. An *AccessDenied* error is returned to the second user.
2) The fork may already be open with a Deny Mode that prohibits the second user's desired access. For example, the first user opened the fork with Deny Mode indicating *Deny Write*, and a second user tries to open the fork for *Write*. A *DenyConflict* error is returned to the second user.
3) The fork may already be open with a Deny Mode that conflicts with the second user's desired Deny Mode. For example, the first user opened the fork for *Write* and *Deny None*. The second user tries to open the fork with Deny Mode indicating *Deny Write*. This request cannot be granted since the fork is already open for *Write*. A *DenyConflict* error is returned to the second user.

Deny Modes are cumulative in that each successful open of a fork "combines" its Deny Mode with previous Deny Modes. That is, if the first open sets a Deny Mode of *Deny Read* and the second sets a Deny Mode of *Deny Write*, the fork's Current Deny Mode (CDM) will be *Deny Read/Write*. *Deny None* and *Deny Read* combine to form a Current Deny Mode of *Deny Read*. Likewise, Access Modes are cumulative: if the first open gets *Read* access and the second gets *Write* access, the Current Access Mode (CAM) is *Read/Write*.

The rules for allowing or denying multiple opens of a fork depend on the Current Deny Mode, the Current Access Mode, and the Deny and Access Modes being requested in a new *FPOpenFork* call, and are summarized in the table below. A check mark indicates that the new open is allowed, otherwise it is denied.

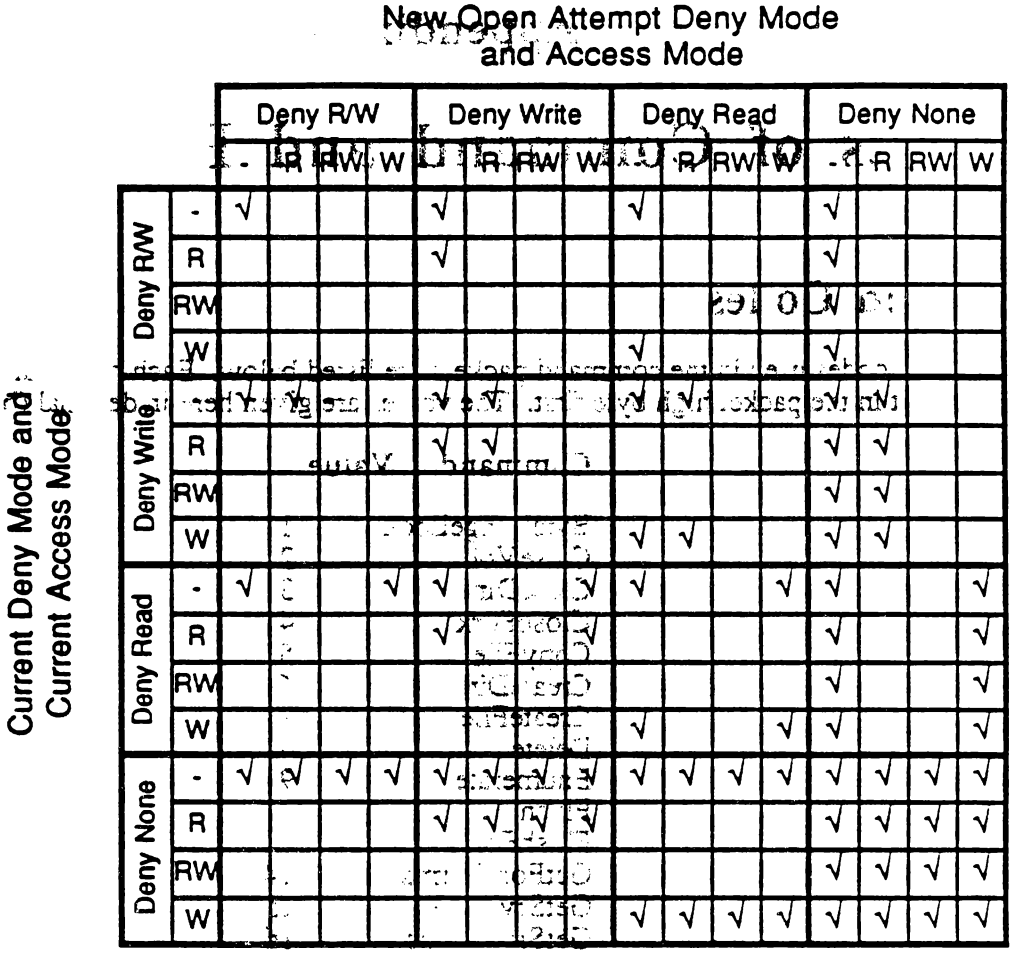


Figure C-1: Rules for Allowing or Denying Multiple Opens

Appendix D

Values of Command and Error Codes

Command Codes

The command codes used in the command packets are listed below. Each command code is a 16 bit integer sent in the packet high byte first. The values are given here in decimal (base 10) form.

Command	Value
ByteRangeLock	1
CloseVol	2
CloseDir	3
CloseFork	4
CopyFile	5
CreateDir	6
CreateFile	7
Delete	8
Enumerate	9
Flush	10
FlushFork	11
GetForkParms	14
GetSrvrInfo	15
GetSrvrParms	16
GetVolParms	17
Login	18
LoginCont	19
Logout	20
MapID	21
MapName	22
Move	23
OpenVol	24
OpenDir	25
OpenFork	26
Read	27
Rename	28
SetDirParms	29
SetFileParms	30
SetForkParms	31
SetVolParms	32
Write	33
GetFileDirParms	34
SetFileDirParms	35
OpenDT	48
CloseDT	49
GetIcon	51

Error Codes

Each call returns an Error code which is a 4-byte integer. The various error values are listed below together with their mnemonic names (these are the names used in Chapter 7). The values given below are in hexadecimal and decimal (base 10) form.

Error Mnemonic	Hex Value	Decimal Value
NoErr	$0	0
AccessDenied	$FFFFEC78	-5000
AuthContinue	$FFFFEC77	-5001
BadUAM	$FFFFEC76	-5002
BadVersNum	$FFFFEC75	-5003
BitmapErr	$FFFFEC74	-5004
CantMove	$FFFFEC73	-5005
DenyConflict	$FFFFEC72	-5006
DirNotEmpty	$FFFFEC71	-5007
DiskFull	$FFFFEC70	-5008
EOFErr	$FFFFEC6F	-5009
FileBusy	$FFFFEC6E	-5010
FlatVol	$FFFFEC6D	-5011
ItemNotFound	$FFFFEC6C	-5012
LockErr	$FFFFEC6B	-5013
MiscErr	$FFFFEC6A	-5014
NoMoreLocks	$FFFFEC69	-5015
NoServer	$FFFFEC68	-5016
ObjectExists	$FFFFEC67	-5017
ObjectNotFound	$FFFFEC66	-5018
ParamErr	$FFFFEC65	-5019
RangeNotLocked	$FFFFEC64	-5020
RangeOverlap	$FFFFEC63	-5021
SessClosed	$FFFFEC62	-5022
UserNotAuth	$FFFFEC61	-5023
CallNotSupported	$FFFFEC60	-5024
ObjectTypeErr	$FFFFEC5F	-5025
TooManyFilesOpen	$FFFFEC5E	-5026
ServerGoingDown	$FFFFEC5D	-5027
CantRename	$FFFFEC5C	-5028
DirNotFound	$FFFFEC5B	-5029
IconTypeError	$FFFFEC5A	-5030

Appendix E
List of Abbreviations

AFI	AppleTalk filing interface
AFP	AppleTalk Filing Protocol
ALAP	AppleTalk Link Access Protocol
ASP	AppleTalk Session Protocol
ATP	AppleTalk Transaction Protocol
CAM	Current Access Mode
CDM	Current Deny Mode
DDP	Datagram Delivery Protocol
GMT	Greenwich Mean Time
HFS	Macintosh's hierarchical file system
MFS	the earlier Macintosh (flat) file system
NBP	Name Binding Protocol
NFI	Native filing interface
SLS	Session listening socket
UAM	User authentication method